VISOKO UČILIŠTE ALGEBRA

ZAVRŠNI RAD

Prikaz i pokret 3D modela u prostoru

Aron Thompson

Zagreb, veljača 2019.

„Pod punom odgovornošću pismeno potvrđujem da je ovo moj autorski rad čiji niti jedan dio nije nastao kopiranjem ili plagiranjem tuđeg sadržaja. Prilikom izrade rada koristio sam tuđe materijale navedene u popisu literature, ali nisam kopirao niti jedan njihov dio, osim citata za koje sam naveo autora i izvor, te ih jasno označio znakovima navodnika. U slučaju da se u bilo kojem trenutku dokaže suprotno, spreman sam snositi sve posljedice uključivo i poništenje javne isprave stečene dijelom i na temelju ovoga rada".

U Zagrebu, 18.2.2019.

Aron Thompson

Predgovor

Ovom prilikom bih se htio zahvaliti obitelji koja ne samo da je bila prisiljena slušati o mom radu, nego je i sudjelovala u nekim dijelovima, Alenu Ladavcu iz Croteam-a na ideji za rad i mentoru Goranu Đambiću za brze odgovore na moje upite, savjete za rad kao i odlične recenzije rada.

Prilikom uvezivanja rada ne zaboravite umjesto ove stranice umetnuti original potvrde o prihvaćanju teme završnog rada koji ste preuzeli u studentskoj referadi

Sažetak

Ideja ovog rada je predstaviti i pojasniti 3D računalnu grafiku, te uisto izraditi baznu aplikaciju koja se može lako nadograditi i dodati osobnom portfelju koji je poželjan za zapošljavanje u tipu industrije kojem su potrebna takva znanja.

Programeru je za rad u tvrtkama koje izrađuju prestižne igre potrebno je dubinsko znanje računalne grafike budući da mnogi koriste vlastite okvire ili posebno izrađene module za izradu igara. Kada se radi na najmodernijoj grafici, koriste se najnovije tehnike i često je potrebno izraditi aplikacije i *shadere* specifične za takve projekte. Cilj rada je upoznavanje s osnovama OpenGL aplikacijskog programskog sučelja (engl. *Application Programming Interface*, skraćeno API), njegovog grafičkog cjevovoda i biblioteke usko vezane za rad s OpenGL-om. Matematički modeli su neizbježni za rad s 3D grafikom pa će se primjerima demonstrirati osnove 3D matematike. Pojasnit će se i izraditi *shaderi* koji daju boju i nijanse modelima, a za modele će se omogućiti transformacije. Također će se prikazati obilazak kroz prostor izradom kamere. Sve će se popratiti izradom 3D grafičke aplikacije kojom će se omogućiti kretanje modela u 3D prostoru. Aplikacija će služiti kao baza koja omogućava jednostavnu nadogradnju i izradu aplikacije za portfelj.

OpenGL je svestran i zato se puno koristi. Iako polako zastarijeva i ima slabije performanse u usporedbi s nekim drugim grafičkim programskim sučeljima, primjenjivat će se još dulji period vremena.

Ključne riječi: 3D računalna grafika, Open Graphics Library API, matematika, *shaderi*, transformacije, kamera, portfelj.

Summary

The idea of this work is to present and demonstrate 3D computer graphics and build a base of application that can be easily upgraded and used for personal application portfolio recommended for seeking a job in that type of industry.

For a programmer working at a high-end game development company, a deep knowledge of computer graphics is necessary due to many custom modules and company-based game engines. When newest graphics techniques are used, it is often necessary to create applications and shaders specific to those projects. For those reasons this work will demonstrate basics of Open Graphics Library API and it’s graphics pipeline, as well as it's commonly used libraries. Mathematics unavoidable for 3D graphics will be demonstrated through examples. Shaders for colouring and toning of models will be explained and demonstrated. Transformations will be applied on models and movement through space will be enabled. Both of those functionalities will be explained through theory and code. Theory will be enhanced with code examples for creation of a base functional 3D graphics application, which can easily get enhanced and used for portfolio.

OpenGL is widely used a lot because of its versatility. Although it is getting outdated and other graphical APIs have been outperforming it, it will still be in use for a long time.

Keywords: 3D computer graphics, Open Graphics Library, mathematics, shaders, transformations, camera, portfolio.

Sadržaj

1. Uvod

Glavna ideja ovog projekta je izrada portfelja potrebnog za zapošljavanje u visokokvalitetnim tvrtkama koje se bave razvojem računalnih igara i sličnim područjima. Za temu rada sam se odlučio nakon savjetovanja s glavnim programerom jedne takve hrvatske tvrtke.

Ovaj projekt i rad predstavljaju uvod u rad s 3D grafikom općenito, ali su primjenjivi i za rad s igrama. Sva osnovna područja će se pokriti i pojasniti primjerima što će dati dobar uvid u potrebnu vrstu znanja i vještina koje se trebaju razvijati za takav tip karijere.

Rad se bazira na tehnologiji Open Graphics Library (skraćeno OpenGL) API-a. Predstavit će se sam OpenGL, kao i biblioteke koje su potrebne za učinkovit rad s OpenGL-om, jer OpenGL-u nedostaju neke osnovne mogućnosti, poput izrade prozora. Matematika je izuzetno zastupljena u radu s 3D grafikom, tako da će se napraviti uvodni pregled matematike, konkretno vektora i matrica, a kasnije će se opisivati specifične matematičke tehnike po potrebi.

Nakon uvodne matematike krenut će se s izradom *shadera*, što označava početak rada s grafikom i iscrtavanjem modela. Od funkcionalnosti će se prvo prikazati transformacije modela u 3D prostoru, i to na sve načine: od translacije, preko rotacije, do skaliranja. Zatim će se izraditi i demonstrirati simulacija kamere koja će omogućiti slobodno kretanje kroz prostor. U predzadnjem poglavlju će se opisati slaganje aplikacije, a u zadnjem poglavlju rada će se usporediti OpenGL API s drugim popularnim grafičkim API-ima te će se prikazati njihove performanse, karakteristike, prednosti i nedostaci.

Glavni izvori rada su redom knjige [2] i [1].

2. Motivacija, rad i perspektiva rada

Važan dio života je karijera. Razvoj računalnih igara je vrlo privlačna grana programiranja koja je možda najbliža umjetnosti. Upravo zato je izuzetno kompetitivna pa se između ostaloga znaju čuti upozorenja pri ulasku u tu industriju o većem broju radnih sati ili nižem primanjima nego u drugim granama programiranja. Ako vam je razvoj računalnih igara zanimljiv i potiče vas da naučite nešto više i napravite nešto bolje te želite velik dio života provesti u poslu koji vas ispunjava, čvrsto vjerujem da se isplati ostvariti karijeru na tom području. Kroz daljnji tekst poglavlja će se detaljnije objasniti moja osobna motivacija za rad i budući koraci prema željenoj karijeri.

2.1. Motivacija

Motivacija za ovaj rad dolazi iz potrebe i želje za usmjeravanjem karijere. Nakon boljeg upoznavanja s mogućnostima i vrstama programiranja tijekom studija, shvatio sam koji bi me logični koraci mogli dovesti do željenog cilja. Dugo nisam imao konkretan plan, dobrim dijelom zbog nepoznavanje struke i tržišta. Oduvijek sam uživao u računalnim igrama svih vrsta, i kao igrač i u njihovoj analizi. U jednom sam trenutku tijekom druge godine na Visokom učilištu Algebra počeo ozbiljno razmatrati opciju takve karijere te se počeo raspitivati hoće li postojati kolegiji vezani za razvoj igara. Ubrzo nakon toga je objavljen diplomski studij razvoja računalnih igara i to je bio trenutak kada sam definitivno donio odluku: to će područje biti moj karijerni cilj.

Znao sam bih se najviše želio baviti izradom vrhunskih 3D igara, pa sam na savjet mentora kontaktirao poznatu hrvatsku tvrtku koja se time bavi i ima puno proizvedenih uspješnih igara iza sebe. Navedena se firma također bavi izradom igara virtualne stvarnosti,[1] koje su vjerojatnost budućnost igara, što je samo povećalo moj interes. Cilj mi je jednoga dana raditi u takvom okruženju pa sam od njih zatražio pomoć pri odabiru teme za izradu završnog rada kako bih izgradio početna znanja potrebna za eventualni ulazak u takvu industriju. Na moj upit je odgovorio njihov glavni tehnički direktor (engl. *chief technology officer,* skraćeno CTO), i poslao primjere radova zaposlenika koji su im pomogli pri zapošljavanju, te mi dao smjernice za rad. Par mjeseci kasnije, u travnju 2018. otputovao sam u Dubrovnik na

[1] http://www.croteam.com/serious-sam-vr-last-hope/

konferenciju za razvoj igara, Reboot Develop, koji se od 2019. godine zove Reboot Develop Blue[2]. Tamo sam se bolje upoznao s raznim područjima i samom industrijom razvoja igara kroz predavanja i druženja, a upoznao sam i CTO-a koji mi je ukratko rekao da je u Hrvatskoj jedina prava opcija samostalno naučiti i sakupiti znanja potrebna za rad na razini potrebnih za sudjelovanje u razvoju vrhunskih igara.

2.2. Rad i perspektiva rada

Kroz rad će se izraditi iscrtavanje i transformacija modela te će se omogućiti kretanje kroz prostor koristeći tehniku kamere. Koristit će se biblioteke Open Asset Import Library (skraćeno Assimp) za olakšavanje učitavanja 3D modela, OpenGL Mathematics (skraćeno GLM) za jednostavnije baratanje matematikom, Graphics Library Framework (skraćeno GLFW) za izradu prozora u kojem će se iscrtati 3D modeli, OpenGL Extension Wrangler Library (skraćeno GLEW) za učitavanje ekstenzija i Open Graphics Library sučelje za programiranje kojim se obavljaju pozivi prema grafičkom upravljačkom programu (engl. *graphic driver*). Pozivi prema grafičkom upravljačkom programu su najniži pozivi koji se obavljaju za komuniciranje s grafičkom karticom, osim prilikom izrade samog grafičkog upravljačkog programa, ali njega izrađuju proizvođači same kartice.

OpenGL potiče korištenje raznih biblioteka, jer sam po sebi ne pokriva sve potrebe za izradu grafičkih programa. Biblioteke pomažu izradi aplikacije, ali cijela logika kretanja, iscrtavanja, osvjetljenja, kamere i svih budućih funkcionalnosti se moraju samostalno posložiti i kodirati u jeziku C++. Izazovno je jer ne postoji prava kamera, kretanje, niti išta slično po pitanju funkcionalnosti, već je potrebno složiti njihovu simulaciju, kao i simulaciju 3D prostora, a sve to je zapravo matematička obrada koja omogućuje prikaz na 2D ekranu. Ukratko, sve funkcionalnosti je potrebno izraditi, a za većinu su potrebni matematički izračuni. Potrebno je izraditi i *shadere,* male programe koji se pokreću veliki broj puta na grafičkoj kartici, koristeći OpenGL shading language (skraćeno GLSL). Pomoću njih se radi skoro sva obrada grafike.

Na tržištu postoje okviri za izradu igara, poput Unreal Engine i Unity Engine okvira koji rade sve to i puno više. Razlog zašto je znanje o detaljnom radu grafike korisno je to što većina velikih tvrtki koje se bave razvojem igara imaju ili svoj okvir za izradu igara ili rade dodatne module za okvire. Znanje u takvom području je također korisno prilikom izrade

[2] http://rebootdevelopblue.com/

vodećih igara industrije, jer se za njih radi s najnovijim tehnologijama i funkcionalnostima, što zahtijeva izradu posebnih modula koji će te najnovije i najimpresivnije mogućnosti realizirati prije konkurencije. Također je poželjno imati okvir za razvoj igara koji odgovara tipu igara koji se razvija, jer niti jedan okvir nije najbolji u svemu, što znači da mnoge tvrtke stalno razvijaju svoj. Croteam koristi Serious Engine[3], Ubisoft koristi AnvilNext[4], Activision koristi IW engine[5], Crytek koristi CryEngine[6], a to su samo neki od primjera koji pokazuju koliko je takvo znanje važno i potrebno.

Ovaj rad predstavlja uvod i osnove rada s 3D grafikom u OpenGL-u, ali daje i dobar pregled rada s 3D grafikom općenito. Sam rad će se dalje razvijati u dva smjera. Jedan je ugradnja u vlastiti okvir za razvoj igara, a drugi su daljnja poboljšanja i funkcionalnosti prikaza grafike. Dodat će se potpuno Phong osvjetljenje (engl. *Phong shading*) [7], sjene, geometrijski *shader*, okluzija okoline[8], a kasnije ako, bude moguće, *antialiasing* i anizitropno filtriranje (engl. *anisitropic filtering*, skraćeno AF*).* Takva znanja su najčešće dostatna za početak profesionalne karijere u kvalitetnim tvrtkama.

[3] http://www.croteam.com/technology/
[4] https://en.wikipedia.org/wiki/AnvilNext/
[5] https://en.wikipedia.org/wiki/IW_engine/
[6] https://en.wikipedia.org/wiki/CryEngine/
[7] https://developer.nvidia.com/vxao-voxel-ambient-occlusion
[8] https://en.wikipedia.org/wiki/Phong_shading

3. Osnove OpenGL sučelja i ključnih biblioteka

OpenGL je API za programiranje raznih 2D i 3D grafičkih aplikacija. Za izradu aplikacija koje koriste OpenGL redovito se koriste ekstenzije i biblioteke koje olakšavaju rad s OpenGLom i dodaju mu nove mogućnosti. S obzirom da se neke od tih biblioteka i ekstenzija učestalo koriste, napravit će se pregled nekih najbitnijih. Biblioteke i ekstenzije ne treba izbjegavati jer i konzorcij Khronos group koji održava OpenGL potiče korištenje ekstenzija i biblioteka kao standardnu praksu pri korištenju OpenGL-a.[9]

3.1. Pregled Open Graphics Library-a

Open Graphics Library ili OpenGL je definiran kao sučelje, ali je zapravo samo specifikacija koja specificira kako svaka funkcija treba raditi, dok se implementacija i izrada biblioteka prepušta razvojnim programerima koji najčešće rade za proizvođače grafičkog hardvera. OpenGL je implementiran kao klijent-poslužitelj sustav. Aplikacija koja se piše je klijent, a implementacija grafičkog hardvera je poslužitelj. Iz tog razloga se OpenGL zna ponašati različito ovisno o hardveru prisutnom na računalu, jer za dio o implementaciji je odgovoran proizvođač grafičke kartice.

OpenGL sadrži preko 500 različitih naredbi za korištenje i izradu objekata i slika u dvodimenzionalnim i trodimenzionalnim aplikacijama. Zamišljen je kao efikasno sučelje koje se može implementirati na velik broj različitog grafičkog hardvera, a ako on nije prisutan, može se implementirati i u softveru. Neovisan je o operativom sustavu računala, ali zbog toga OpenGL ne uključuje funkcije za obradu korisničkog unosa, izradu prozora u kojem će se program prikazivati, opisivanje dvodimenzionalnih i trodimenzionalnih objekata, kao niti mogućnost učitavanja vanjskih datoteka. Te se funkcionalnosti omogućavaju korištenjem već gotovih biblioteka ili izradom vlastitih.

OpenGL je objavljen u lipnju 1992. kao verzija 1.0. Uključujući prvu verziju, do danas je izašlo 20 verzija. OpenGL zadržava kompatibilnost unatrag, ali budući da to stvara značajne troškove zbog velikih promjena i razlika u hardveru i softverskim praksama, ono što je prije dobro funkcioniralo danas je zastarjelo i teško za održavanje. Upravo zbog toga je ARB, današnji Khronos group napravio podjelu na dva glavna profila, osnovni moderni za moderni

[9] https://www.khronos.org/opengl/wiki/OpenGL_Extension#Core_Extensions

hardver i profil kompatibilnosti koji radi sa svim verzijama unatrag do verzije 1.0. Moderni profil kreće s verzijom 3.0 i trenutno ide do 4.6 koji je izašao 2017. godine.[6]

3.2. OpenGL Extension Wrangler Library

OpenGL extension wrangler library (skraćeno GLEW) je višeplatformska C i C++ biblioteka otvorenog kôda za učitavanje ekstenzija. GLEW omogućava provjeru podrške ekstenzija na ciljanoj platformi za vrijeme izvođenja kôda.[10]

Sve OpenGL ekstenzije su izložene u jednoj zaglavnoj (engl. *header*) datoteci, koja se automatski generira sa službenog popisa proširenja. GLEW je dostupan na mnogim operativnim sustavima, od kojih su poznatiji Windows, Linux i macOS X.[11] GLEW se ponaša kao sučelje prema OpenGL-u stvarajući funkcije za komunikaciju s važnim novim mogućnostima OpenGL-a korištenjem pokazivača te provjerava postoje li te funkcije na platformi koja se koristi. Korištenje s Visual Studio radnom okolinom je jednostavno. Nakon što se dodaju potrebne datoteke i spoje preko Visual Studio povezivača (engl. *linker*) potrebno je uključiti *header* datoteku, i osnovni kôd za inicijalizaciju, vidljivo kroz **Pogreška! Izvor reference nije pronađen.**

3.3. Graphics Library Framework

Graphics Library Framework (skraćeno GLFW*)* je biblioteka napisana u programskom jeziku C koja omogućava programerima mogućnost stvaranja i upravljanja prozorima kao i obradu unosa preko miša, tipkovnice i kontrolera. Od operativnih sustava podržava Windows, macOS, Linux i mnoge druge. Potreban je stoga što OpenGL ne pruža takve mogućnosti. GLFW nije jedina opcija za rad s prozorima i obradu unosa. Jedna od alternativa je Simple DirectMedia Library (skraćeno SDL), moćnija i svestranija biblioteka koja radi slično kao GLFW, ali nije primarno fokusirana na OpenGL i vrlo je opširna. [12]

[10] http://glew.sourceforge.net/
[11] https://en.wikipedia.org/wiki/OpenGL_Extension_Wrangler_Library
[12] https://en.wikipedia.org/wiki/GLFW

3.4. OpenGL Mathematics

OpenGL Mathematics (skraćeno GLM) je matematička C++ biblioteka za grafička rješenja. Bazirana je na OpenGL Shading Language-u (skraćeno GLSL), jeziku u kojem se programiraju *shaderi*. Klase i funkcije u GLM-u su istog naziva i funkcionalnosti kao i u GLSL-u, što znači da, ukoliko programer zna jezik GLSL, znat će također koristiti i GLM u C++ jeziku. GLM je po pitanju matematike širi od GLSL-a, što omogućava razne izračune poput transformacije matrica, kvaterniona, slučajnih brojeva i mnoge druge. Ova biblioteka je izrađena da bude visoko kompatibilna s OpenGL-om što pomaže da se izrada kôda za računanje svede na minimum.[8] GLM također osigurava interoperabilnost s raznim kompletima za razvoj softvera (engl. *software development kit*, skraćeno SDK) i bibliotekama. Kada se koristi GLM znatno se smanjuje potrebna količina kôda za matematičke izračune. Kôd 3.1 prikazuje pozivanje GLM funkcija.

```
#include <glm/vec3.hpp> // glm::vec3
#include <glm/vec4.hpp> // glm::vec4
#include <glm/mat4x4.hpp> // glm::mat4
#include <glm/gtc/matrix_transform.hpp> // glm::translate,
glm::rotate, glm::scale, glm::perspective
glm::mat4 camera(float Translate, glm::vec2 const & Rotate)
{
      glm::mat4 Projection =
      glm::perspective(glm::radians(45.0f), 4.0f / 3.0f,
0.1f, 100.f);
      glm::mat4 View = glm::translate(glm::mat4(1.0f),
      glm::vec3(0.0f, 0.0f, -Translate));
      View = glm::rotate(View, Rotate.y, glm::vec3(-1.0f,
0.0f, 0.0f));
      glm::mat4 Model = glm::scale(glm::mat4(1.0f),
      glm::vec3(0.5f));
      return Projection * View * Model;
}
```

Kôd 3.1 Primjeri GLM funkcija[13]

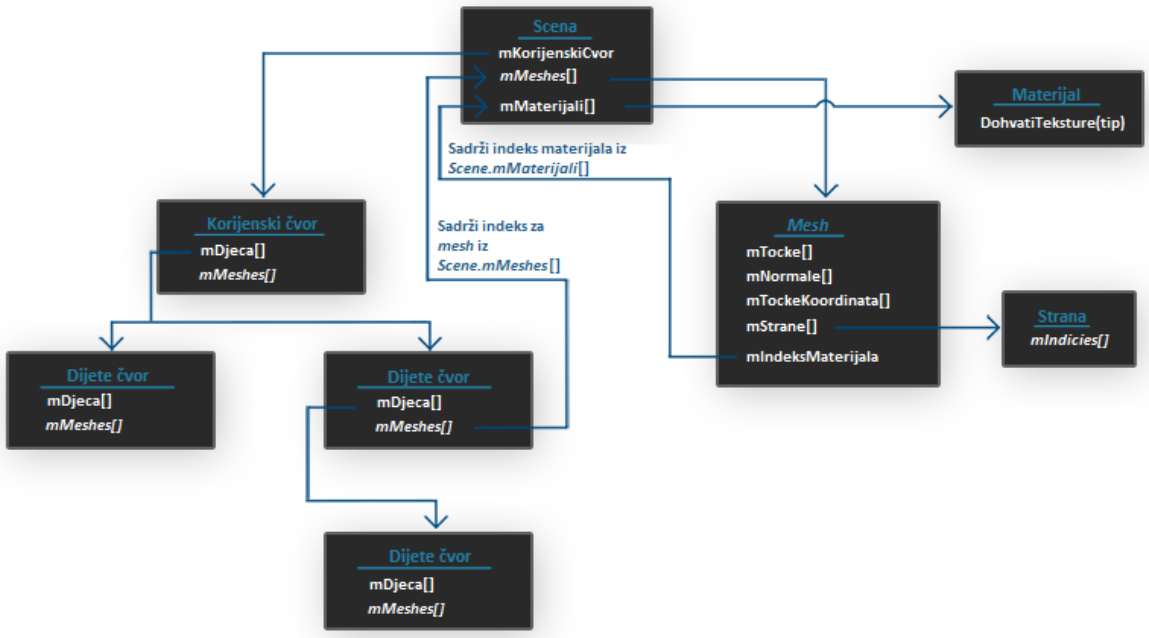
[13] https://glm.g-truc.net/0.9.9/index.html

3.5. Open Asset Import Library

Open Asset Import Library (skraćeno Assimp) je jedna od biblioteka za uvoz modela iz datoteka. Može uvesti desetke različitih formata datoteka s modelima. Učitava ih u svoje opće strukture podataka. Format modela iz datoteke se sakrije iza sučelja, što omogućava jedinstveno ponašanje biblioteke bez obzira na format.

Assimp učitava datoteku tako da cijeli model spremi u objekt koji se zove scena (engl. *scene*). Scena sadrži sve podatke tog modela. Assimp biblioteka ima pristup kolekciji čvorova. Čvorovi imaju spremljene podatke, a svaki čvor može imati još čvorova, svoju djecu čvorove.

Prvo je potrebno učitati objekt scene, zatim rekurzivno dohvatiti odgovarajuće mreže objekta (engl. *mesh*) iz čvorova te njihovu djecu. Iz svakog *mesha* se dohvaćaju podaci o točci, vrhu i materijalu. Dobiveni podaci se spremaju u objekt model. Hijerarhija podataka je prikazana na Slika 3.1.



Slika 3.1 Hijerarhija podataka u biblioteci Assimp[14]

Opis podatkovnih modela u Assimpu:

- osnovni objekt je Scena

[14] https://learnopengl.com/Model-Loading/Assimp

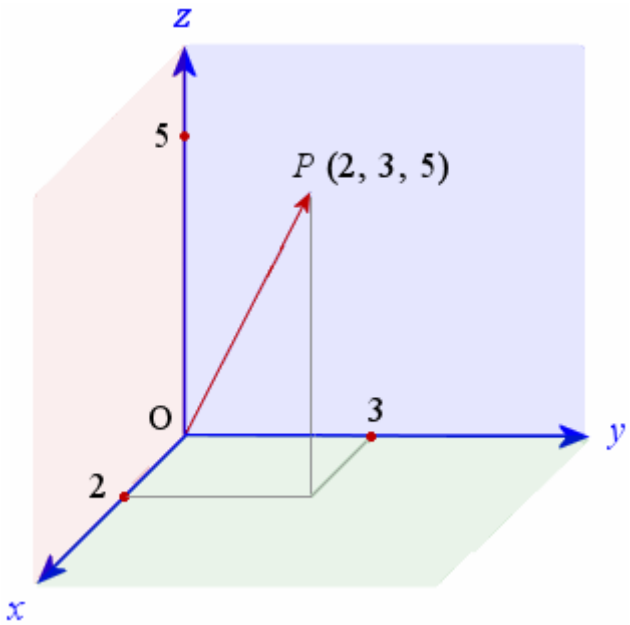
- iz njega se učitava mKorijenskiCvor koji sadrži *mesh*, a može sadržavati i djecu čvorove koji također imaju *mesheve*, a mogu imati i djecu čvorove
- *mesh* objekt sadrži podatke potrebne za iscrtavanje: pozicije vrhova (engl. *vertex position*), normale vektora i koordinate teksture, strane i indeks materijala
- *mesh* može sadržavati više strana
- Materijal sadrži funkcije koje dohvaćaju vrijednosti poput boja i teksture od npr. osvjetljenja

4. Pregled osnovnih područja matematike za računalnu grafiku

Vektori i matrice su izuzetno važni za rad s 3D grafikom. Njima se predstavljaju točke u prostoru koje služe kao prikaz lokacije objekata u igrama. Potrebni su, između ostaloga, za prikaz smjera, orijentaciju kamere, normale površine objekata, promjenu oblika i pozicije objekta. Razumijevanje manipulacije vektorima i matricama je osnovna vještina programera 3D grafike.

4.1. Vektori

Vektori su matematičke i fizikalne veličine koje su određene s više od jednog podatka. Vektori mogu biti elementi jednodimenzionalnih, dvodimenzionalnih, trodimenzionalnih i općenito *n*-dimenzionalnih prostora. Za rad s grafikom se koriste vektori u 3D prostoru. Vektori 3D prostora su određeni duljinom i smjerom.



Slika 4.1 Vektor OP u 3D prostoru[15]

Na Slika 4.1 može se vidjeti da je početna točka vektora OP O (0, 0, 0) te da je krajnja točka P (2, 3, 5). Za OpenGL i 3D grafiku su najvažnije točke u prostoru, a kako su točke i vektori

[15] https://www.intmath.com/vectors/7-vectors-in-3d-space.php

ekvivalentni, razmatranjem vektora se može puno opisati, izračunati i zaključiti. Točka predstavlja poziciju u prostoru, preciznije, označava x, y i z u trodimenzionalnom prostoru.

Nadalje ćemo se baviti točkama kao vektorima. Vektori se mogu zapisati na više načina.

Najčešći zapisi su u obliku uređene trojke brojeva (1) ili u obliku matrice (2) u jednom stupcu. Matrični zapis je pogodniji za računanje u 3D grafici.

$$\mathrm{P} = \left(\mathrm{P_x}, \mathrm{P}_y, \mathrm{P}_z\right) = (2, 3, 5) \tag{1}$$

$$\mathrm{P} = \begin{bmatrix} \mathrm{P_x} \\ \mathrm{P}_y \\ \mathrm{P}_z \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 5 \end{bmatrix} \tag{2}$$

4.1.1. Zbrajanje i oduzimanje vektora

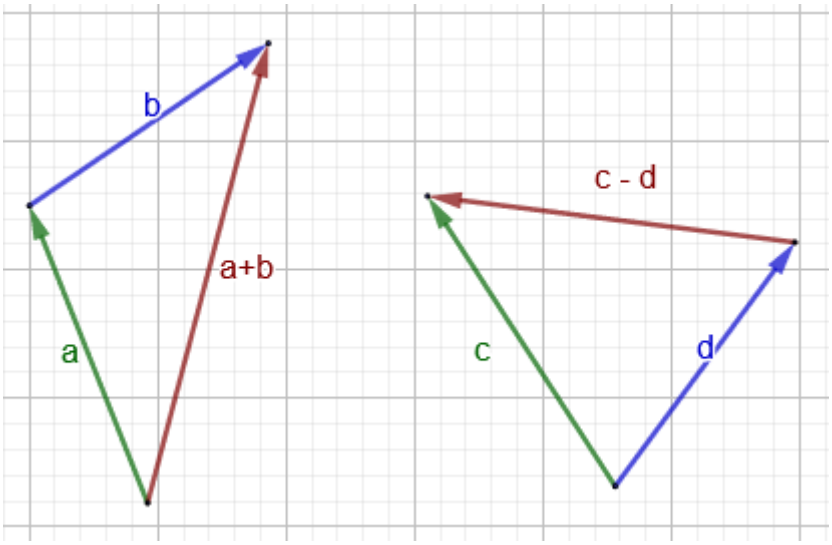
Zbrajanje i oduzimanje trodimenzionalnih vektora se vrši tako što se u okomitom zapisu vektora zbrajaju ili oduzimaju vrijednosti istih redaka. Za zbrajanje i oduzimanje broj elemenata u vektorima mora biti identičan. To se može vidjeti iz jednostavne formule za zbrajanje vektora (3) i primjera zadatka s rješenjem (4), te formule (5) i primjera zadatka (6) za oduzimanje vektora. Na Slika 4.2 se može vidjeti kako zbrajanje i oduzimanje vektora izgleda u prostoru.

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} + \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} x_1 & + & x_2 \\ y_1 & + & x_2 \\ z_1 & + & z_2 \end{bmatrix} \tag{3}$$

$$\begin{bmatrix} 2 \\ 3 \\ 5 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 4 \end{bmatrix} = \begin{bmatrix} 2 & + & 1 \\ 3 & + & 2 \\ 5 & + & 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 5 \\ 9 \end{bmatrix} \tag{4}$$

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} - \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} x_1 & - & x_2 \\ y_1 & - & x_2 \\ z_1 & - & z_2 \end{bmatrix} \tag{5}$$

$$\begin{bmatrix} 2 \\ 3 \\ 5 \end{bmatrix} - \begin{bmatrix} 1 \\ 2 \\ 4 \end{bmatrix} = \begin{bmatrix} 2 & - & 1 \\ 3 & - & 2 \\ 5 & - & 4 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \tag{6}$$



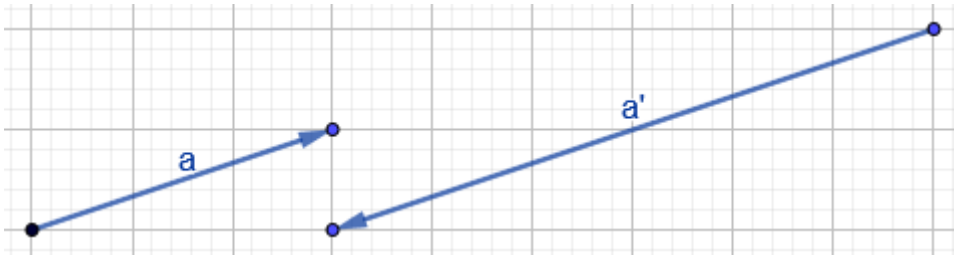
Slika 4.2 Grafički prikaz zbrajanja i oduzimanja vektora

4.1.2. Množenje vektora sa skalarom

Množenje vektora sa skalarom znači pomnožiti svaki element vektora sa skalarom. To je vidljivo iz formule (7) i primjera (8). Pri množenju s negativnim skalarom vektor mijenja smjer. Takav slučaj je vidljiv na Slika 4.3. Vektor a je pomnožen sa negativnim skalarom -2.

$$V = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \cdot a = \begin{bmatrix} x \cdot a \\ y \cdot a \\ y \cdot a \end{bmatrix} \tag{7}$$

$$V = \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix} \cdot 2 = \begin{bmatrix} 4 \\ 6 \\ 8 \end{bmatrix} \tag{8}$$



Slika 4.3 Vektor množen negativnim skalarom

4.1.3. Veličina i normalizacija vektora

Veličina vektora je duljina vektora. Može se izračunati varijacijom Pitagorinog poučka (9) (10)[3]. Dobivenu veličinu možemo normalizirati (11). Normalizirati znači pretvoriti duljinu vektora na 1. Normalizacija olakšava mnoge 3D grafičke izračune.

$$\|V\| = \sqrt{V_x^2 + V_y^2 + V_z^2} \tag{9}$$

$$V = \begin{bmatrix} 2 \\ 3 \\ 6 \end{bmatrix} \quad \|V\| = \sqrt{2^2 + 3^2 + 6^2} = 7 \tag{10}$$

$$\hat{V} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \cdot \frac{1}{\|V\|} \tag{11}$$

4.1.4. Skalarni i vektorski produkt vektora

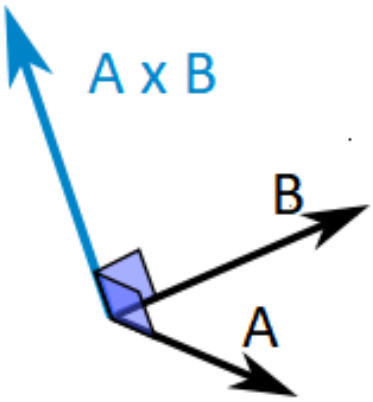
Skalarnim produktom se računa mjera razlike smjerova prema kojima pokazuju dva vektora. Može se izračunati na dva načina. Množenjem elemenata na istim pozicijama u svakom vektoru (12) i umnoškom normaliziranih vektora s kosinusom njihovog kuta (13).

$$A \cdot B \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_n \end{bmatrix} \bullet \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = A_1 \cdot B_1 + A_2 \cdot B_2 + \cdots A_n \cdot B_n \tag{12}$$

$$A \cdot B = \|A\| \cdot \|B\| \cdot cos \propto \tag{13}$$

Vektorski produkt vektora (14) daje novi vektor koji je okomit na oba vektora pomnožena međusobno kao što se vidi na Slika 4.4. Vektorskim produktom se dobivaju normale površine.

$$A \times B = \langle A_y \cdot B_z - A_z \cdot B_y, A_z \cdot B - A_x \cdot B_z, A_x \cdot B_y - A_y \cdot B_x \rangle \tag{14}$$



Slika 4.4 Vektorski produkt[16]

4.2. Matrice

Matrice su sustavi brojeva raspoređenih u jednakim redovima i stupcima. Svi redovi i svi stupci moraju imati isti broj elemenata, ali broj elemenata stupca i retka može biti različit. Matricu možemo smatrati i skupom vektora raspoređenih u retke i stupce. Matrice se mogu množiti i zbrajati međusobno, a s vektorima i skalarima se mogu množiti. [2]

Matrice mogu imati različiti broj redaka i stupaca, ali svi retci moraju imati isti broj elemenata. Isto vrijedi za stupce. Primjer 4x3 matrice je prikazan ispod (15).

$$\begin{bmatrix} 2 & 3 & 4 \\ 5 & 1 & 3 \\ 4 & 0 & 4 \\ 1 & 6 & 1 \end{bmatrix} \tag{15}$$

4.2.1. Zbrajanje i oduzimanje matrica

Dimenzije matrica se moraju podudarati jer se zbrajaju (16) i oduzimaju (17) elementi s istih pozicija u različitim matricama.

$$\begin{bmatrix} 2 & 4 \\ 1 & 1 \\ 4 & 3 \end{bmatrix} + \begin{bmatrix} 6 & 0 \\ 4 & 7 \\ 4 & 5 \end{bmatrix} = \begin{bmatrix} 2+6 & 4+0 \\ 1+4 & 1+7 \\ 4+4 & 3+5 \end{bmatrix} = \begin{bmatrix} 8 & 4 \\ 5 & 8 \\ 8 & 9 \end{bmatrix} \tag{16}$$

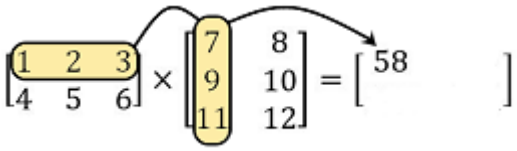
$$\begin{bmatrix} 2 & 4 \\ 1 & 1 \\ 4 & 3 \end{bmatrix} - \begin{bmatrix} 6 & 0 \\ 4 & 7 \\ 4 & 5 \end{bmatrix} = \begin{bmatrix} 2-6 & 4-0 \\ 1-4 & 1-7 \\ 4-4 & 3-5 \end{bmatrix} = \begin{bmatrix} -4 & 4 \\ -3 & -6 \\ 0 & -2 \end{bmatrix} \tag{17}$$

[16] https://www.mathsisfun.com/algebra/vectors-cross-product.html

4.2.2. Množenje matrica

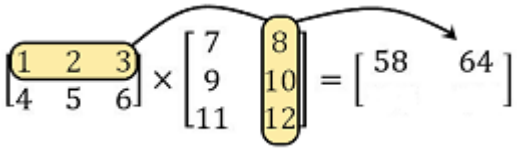
Matrice se mogu množiti samo ako su ulančane, odnosno prva matrica mora imati stupaca koliko druga ima redaka. Elementi rezultantne matrice se dobivaju skalarnim produktom svakog retka sa svakim stupcem.[17]

Rezultat izračuna je matrica koja se dobiva skalarnim produktom svakog retka sa svakim stupcem. Polovica postupka se može vidjeti kroz Slika 4.5, Slika 4.6, i izračune (18), (19), (20) i (21). Rezultat je vidljiv na izračunu (22).



Slika 4.5 Množenje matrica 1[17]

$$\begin{bmatrix}1\\2\\3\end{bmatrix} \bullet \begin{bmatrix}7\\9\\11\end{bmatrix} = 1 \cdot 7 + 2 \cdot 9 + 3 \cdot 11 = 58 \tag{18}$$



Slika 4.6 Množenje matrica 2[17]

$$\begin{bmatrix}1\\2\\3\end{bmatrix} \bullet \begin{bmatrix}8\\10\\12\end{bmatrix} = 1 \cdot 8 + 2 \cdot 10 + 3 \cdot 12 = 64 \tag{19}$$

$$\begin{bmatrix}4\\5\\6\end{bmatrix} \bullet \begin{bmatrix}7\\9\\11\end{bmatrix} = 4 \cdot 7 + 5 \cdot 9 + 6 \cdot 11 = 139 \tag{20}$$

$$\begin{bmatrix}4\\5\\6\end{bmatrix} \bullet \begin{bmatrix}8\\10\\12\end{bmatrix} = 4 \cdot 8 + 5 \cdot 10 + 6 \cdot 12 = 154 \tag{21}$$

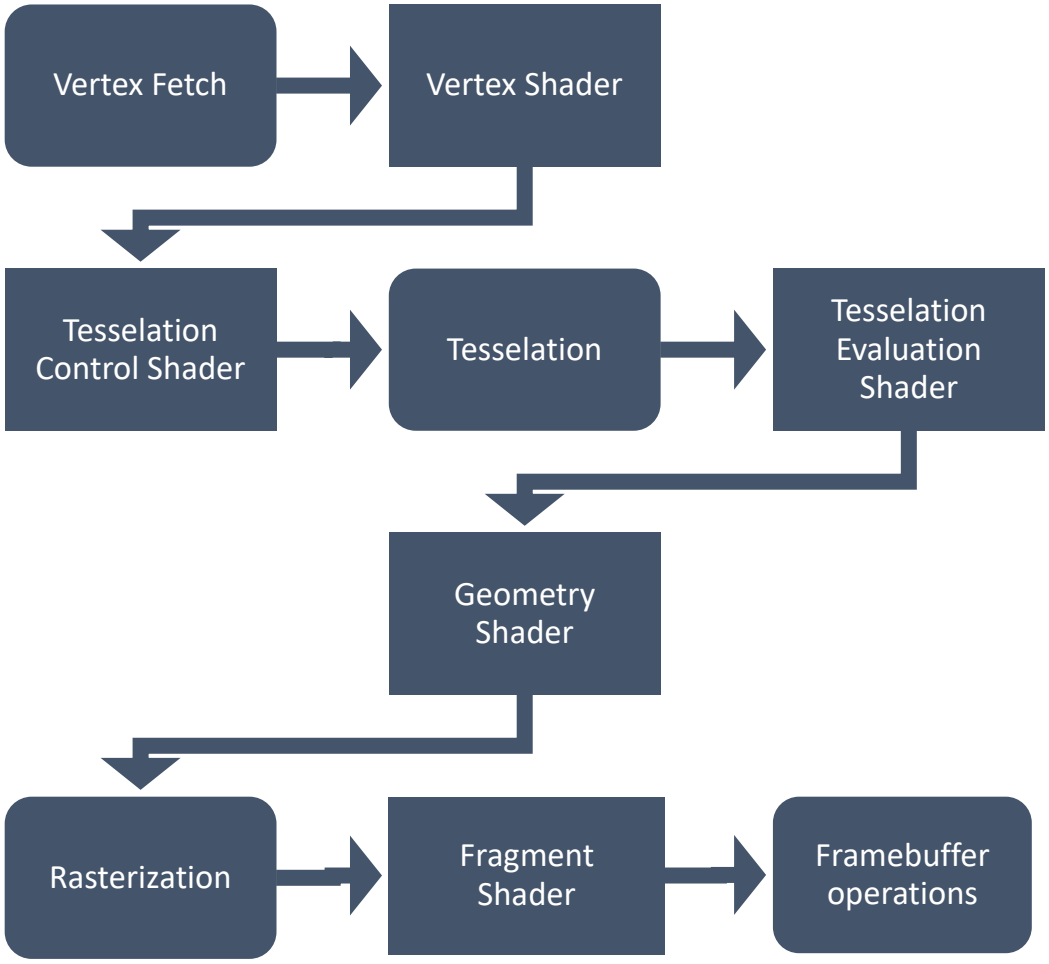
$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} \times \begin{bmatrix}7 & 8\\9 & 10\\11 & 12\end{bmatrix} = \begin{bmatrix}58 & 64\\139 & 154\end{bmatrix} \tag{22}$$

[17] https://www.mathsisfun.com/algebra/matrix-multiplying.html

5. Shaderi i iscrtavanje

Grafičke kartice se sastoje od velikog broja *shader jezgri*, malih i jednostavnih procesora koji se mogu programirati malim programima koji se zovu *shaderi*. Jezgre su jednostavne, ali brojne. Grafičke kartice mogu imati po nekoliko tisuća tih jezgri što omogućuje obradu velike količine podataka [4].

OpenGL grafički cjevovod se sastoji od više faza koje se odvijaju jedna iza druge. Svaka faza predstavlja ili *shader* ili fiksnu funkcija. *Shaderi* se mogu programirati, ali nisu svi obavezni. Slika 5.1 prikazuje pojednostavljeni proces obrade grafike.



Slika 5.1 Pojednostavljeni prikaz faza OpenGL grafičkog sustava [2]

Faze prikazane pravokutnim prozorima se mogu programirati, a one sa zaobljenim ne mogu. Zapravo se sve faze mogu programirati, jer i te faze sadrže kôd, ali njega isporučuje proizvođač grafičkog hardvera preko *drivera* ili *firmewarea*.

Pojednostavljeni opis procesa i glavnih operacija cjevovoda sa **Pogreška! Izvor reference nije pronađen.** prilikom iscrtavanja slike na ekranu:

- Navesti podatke primitiva (vrhovi, linije, trokuti i drugo)
- Odraditi potrebne *shadere* (programe) koji će izračunati položaj, boju i druge atribute primitiva
- Rasterizirati, tj. pretvoriti matematički opis ulaznih primitiva u veći broj manjih dijelova (fragmenata) koji mogu postati pikseli prilikom iscrtavanja
- Izvršiti fragment *shader* za generirane fragmente koji će za svaki fragment odrediti boju i položaj

5.1. OpenGL Shading Language

OpenGL Shading Language (skraćeno GLSL) je baziran na programskom jeziku C po sintaksi, ali je prilagođen za rad s grafikom i paralelan rad općenito. Specifičan je jer ima matrične i vektorske tipove direktno ugrađene u jezik. GLSL je izrađen da radi paralelno na grafičkoj kartici u ogromnom broju kopija [4]. Neke od tih prilagodbi su onemogućena rekurzija i jednostavnija definicija `float` varijable. Ima dobru podršku za skalarne i vektorske tipove i strukture kao i mnoge druge tipove koji su pogodni za rad s teksturama i drugim grafičkim elementima. Važan dio *shadera* su varijable `uniform` kojima OpenGL aplikacija uvijek može pristupiti.

Primjer `uniform` varijabli:

```
uniform float fTime;
uniform vec4 vColorValue;
```

`Uniform` vrijednostima se može pristupiti preko funkcije:

```
glGetUniformLocation(shaderID, "uniformNaziv");
```

Može se pristupiti i preko *layout qualifier*, (u kôdu `layout`) varijable. Ona je specifična za GLSL programski jezik te utječe na mjesto iz kojeg se pristupa podacima vezanim za modele koji će se iscrtati. Prostor za *layout qualifier* varijablu se postavlja koristeći funkciju:

```
void glVertexAttribPointer(  GLuint index, GLint size,
GLenum type, GLboolean normalized, GLsizei stride, const
GLvoid * pointer);
```

`Index` je broj `layouta` kojem se pristupa:

```
layout (location = 2) uniform int iIndex;
```

Ovakvoj `layout` varijabli se pristupa preko `glVertexAttribPointer()` funkcije postavljajući `index` vrijednost na 2. Ovisno o broju varijable se pristupa različitim vrijednostima modela. 1 bi mogao predstavljati x, y i z vrijednosti vrhova modela, a 2 normale modela. Sintaksa pristupa je identična pristupu niza, a vrijednosti su tipa `float`.

Tablica 5.1 prikazuje sve tipove vektora i matrica u GLSL-u.

Dimenzija	Vrsta skalara				
Skalar	bool	float	double	int	unsigned int
2-Element vektor	bvec2	vec2	dvec2	ivec2	uvec2
3-Element vektor	bvec3	vec3	dvec3	ivec3	uvec3
4-Element vektor	bvec4	vec4	dvec4	ivec4	uvec4
2 x 2 matrica	-	mat2	dmat2	-	-
2 x 3 matrica	-	mat2x3	dmat2x3	-	-
2 x 4 matrica	-	mat2x4	dmat2x4	-	-
3 x 2 matrica	-	mat3x2	dmat3x2	-	-
3 x 3 matrica	-	mat3x3	dmat3x3	-	-
3 x 4 matrica	-	mat3x4	dmat3x4	-	-
4 x 2 matrica	-	mat4x2	dmat4x2	-	-
4 x 3 matrica	-	mat4x3	dmat4x3	-	-
4 x 4 matrica	-	mat4x4	dmat4x4	-	-

Tablica 5.1 GLSL vektor i matrični tipovi [2]

OpenGL spaja *shadere*, male programe koji primaju inpute i prosljeđuju upute kroz OpenGL-ove stupnjeve obrade grafike odnosno cjevovod (engl. *pipeline*) sa **Pogreška! Izvor reference nije pronađen.**. *Shaderi* su potrebni za bilo kakvo iscrtavanje. Prevoditelj za GLSL je ugrađen u OpenGL. Kôd *shadera* se sprema u objekt, prevodi, spaja s drugima *shaderima* u program i zatim aktivira. Vrste *shadera* su, *shader* vrha, *shader* mozaične kontrole, *shader* mozaičke procjene, geometrijski *shader* i *shader* fragmenata. *Shader* vrha i fragmenta su osnovni *shaderi* bez kojih se ne može skoro ništa prikazati.

Osnovne OpenGL funkcije za izradu i pokretanje *shadera* su:

- `glCreateShader()` kreira prazni *shader* objekt
- `glShaderSource()` služi za dohvaćanje kôda *shadera* i povezivanje na *shader* objekt
- `glCompileShader()` prevodi kôd *shadera*
- `glCreateProgram()` kreira „program“ objekt na koji se povezuju *shaderi*

- `glAttachShader()` povezuje *shader* na „program” objekt
- `glLinkProgram()` povezuje sve *shader* objekte koji su spojeni na program objekt
- `glUseProgram()` spaja prevedene *shadere* s OpenGL-om i koristi ih za iscrtavanje
- `glDeleteShader()` briše *shader* program nakon što ga OpenGL preuzme

5.2. Shader vrha

Ova faza grafičkog cjevovoda je potpuno programibilna. Odgovorna je za transformacije i osvjetljenje pojedinačnih vrhova. Input ove faze je jedan vrh, ali kako puno *shader* programa radi paralelno, tako se obrađuje veliki broj vrhova paralelno. *Shader* vrha upravlja transformacijom iz prostora modela u prostor pogleda. U praksi, *shaderom* vrha se u 3D grafici rade mnogi napredni procesi poput njihanje lišća i trave na vjetru. Output ovog stupnja je potpuno transformiran i osvjetljen vrh. [9]

Shader vrha u grafičkom cjevovodu radi s geometrijom modela koje korisnik definira, kao i bojama, osvjetljenjem, materijalima, sjenčanjem i teksturom. Iz tih informacija stvara vrhove u prostoru koji imaju vrijednosti boje, dubine, teksture i normale.

Shader vrha zamjenjuje većinu fiksne obrade vrhova, a može mijenjati i koordinate vrhova. Priprema cjelokupno *shader* okruženje za daljnju obradu vrhova kroz *shadere* mozaika, geometrije, zatim za rasterizaciju i na kraju za obradu *shaderom* fragmenata.

5.3. Shader fragmenta

Fragment *shader* je OpenGL faza cjevovoda koja se događa nakon što se primitiv rasterizira, pretvori u manje dijelove, najčešće trokute. Svakom fragmentu koji se dobije rasterizacijom dodjeljuje se skup boja i dubina za iscrtavanje po pikselima (engl. *pixel*). Također sadrži poziciju fragmenta i interpolirane vrijednosti iz faze *shadera* vrha. Vrijednosti izlaza fragment *shadera* su dubina i potencijalno boje koje se mogu zapisati u trenutno aktivne međuspremnike podataka. Fragment *shader* nije obavezan. Glavni izlaz mu je boja tako da ako se postavi samo jedna *output* vrijednost, fragment *shader* će automatski pretpostaviti da je taj izlaz boja te će preuzeti tu vrijednost. Svaki fragment *shader* procesor obrađuje pojedinačni fragment. Varijablama se može pristupiti na isti način kao u *shaderu* vrha.

6. Transformacije modela u 3D prostoru

3D grafika na ekranu se doživljava kao 3D, ali je zapravo iluzija u 2D. Koristimo se 3D konceptima, terminologijom i matematikom da bismo opisali modele koji će se iscrtati u prostoru. Zatim se ti podaci prilagođavaju dodatnom obradom. Uklanja se sve što se ne može vidjeti i prikazuje na 2D ekranu. Proces pretvaranja 3D podataka u 2D se zove projekcija. Projekcija je jedna od transformacija koja se događa u cjevovodu OpenGL-a. Transformacije također omogućuju rotacije, pomicanje i promjenu oblika modela.

6.1. Koordinatni prostor OpenGL-a

Niz transformacija (jedna ili više njih) se mogu zapisati u obliku matrice. Množenjem željenog vektora sa takvom matricom pomičemo taj vektor u prostoru. OpenGL ima nekoliko koordinatnih prostora, a najkorišteniji su prostori modela, pogleda i projekcije. U Tablica 6.1 se mogu vidjeti česte koordinatni prostori 3D grafike.

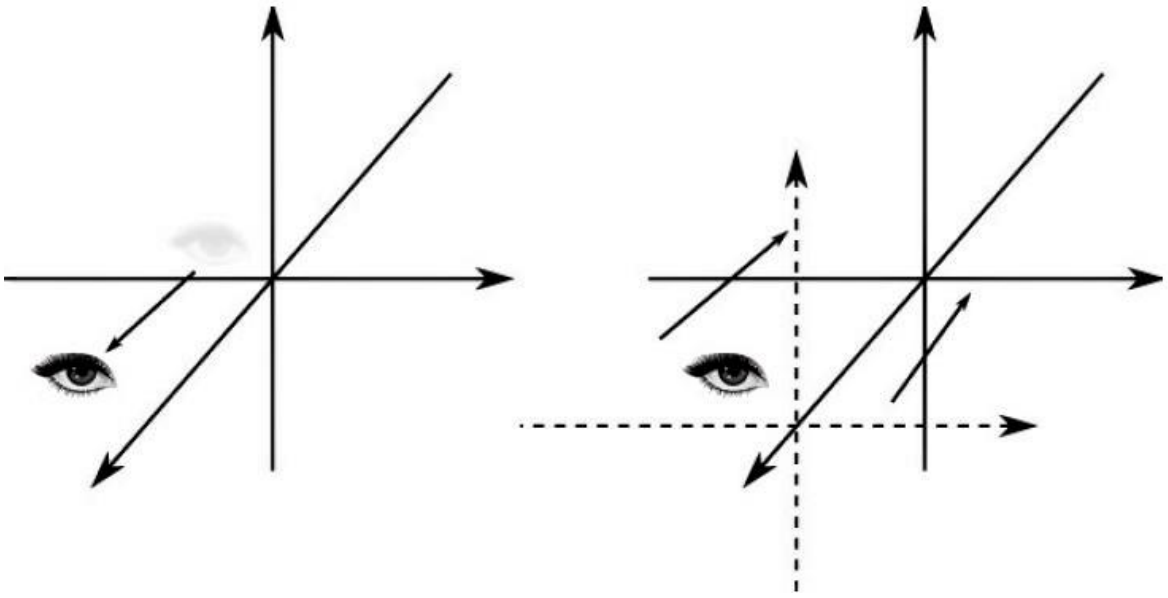
Koordinatni prostor	Što predstavlja
Prostor modela	Pozicija relativna lokalnom ishodištu.
Prostor svijeta	Pozicija relativna globalnom ishodištu.
Prostor pogleda	Pozicija relativna gledatelju. Drugi nazivi su kamera i prostor oka (engl. *eye space)*
Prostor isječka	Pozicije vrhova nakon projekcije u nelinearne homogene koordinate
Prostor prozora	Pozicije vrhova na pikselima relativne prema ishodištu prozora

Tablica 6.1 Česti koordinatni prostori 3D grafičkih aplikacija [2]

Podaci o vrhovima objekta se obično zapisuju u objektni prostor. Pozicije vrhova objekta se računaju s odnosom na jednu početnu poziciju ili točku. Za početnu točku se odabire neka logična točka na objektu kao što je na primjer vrh objekta, tako da rotacijom oko te točke možemo rotirati objekt. Zato se mora odabrati dobra početna točka, jer bi se odabirom točke previše udaljene od predmeta prilikom rotacije oko te iste točke poprilično translatirali taj objekt, a ne samo rotirali. I sa točkom na samom objektu se on translatira, ali ne puno, pretežno se rotira. Problemi znaju nastati kada osoba koja modelira modele stavi ishodište izvan modela pa pomicanje modela postane nepredvidljivo.

Sljedeći koordinatni prostor je prostor svijeta. Koordinate se u njemu računaju u odnosu na fiksnu, globalnu točku, globalno ishodište. Na primjeru svemirskog broda kao našeg objekta, ta točka bi mogla biti neki udaljeni planet. U takvom koordinatnom prostoru svi se objekti računaju u odnosu na istu točku tako da se tu često računa simulacija fizike i osvjetljenje [2].

Koordinatni prostor pogleda su koordinate kamere. To je točka iz kojeg se promatra cijeli naš sustav objekata i transformacija. To su na neki način apsolutne koordinate, koordinate u odnosu na koje se sve ostalo događa, jer kamera je zapravo nepomična i cijeli prostor se okreće oko nje.



Slika 6.1 Perspektive koordinate pogleda [2]

Na Slika 6.1 lijevo se prikazuju koordinate pogleda kako ih vidi promatrač. Desno su koordinate pogleda zarotirane da se bolje vidi odnos osi z. Pozitivne osi x i y su usmjerene desno i gore u odnosu na perspektivu promatrača. Pomicanje u smjeru pozitivne osi z pomiče objekt prema promatraču, a u smjeru negativne osi z odmiče ga od promatrača. To se događa zbog nepomičnosti kamere.

Kako se koordinate premještaju iz jednog koordinatnog prostora u drugi tako se ti vektori množe sa transformacijskim matricama. Tim transformacijama se objekti translatiraju i rotiraju, a oblik im se može promijeniti skaliranjem.

6.2. Translacija, rotacija i skaliranje modela

Za translaciju, rotaciju i skaliranje modela u OpenGL najjednostavnije i najsigurnije je koristiti GLM biblioteku jer štedi puno vremena i kôda. Pogledati ćemo koje se matematičke formule nalaze iza tih transformacija. Kôd 6.1 prikazuje kako se model transformira koristeći GLM biblioteku.

```
glm::mat4 model{ 1.0f };
GLfloat startingAngle = 15.0f;
model = glm::translate(model, glm::vec3(-8.0f, 2.0f,
0.0f));
model = glm::rotate(model, glm::radians(+startingAngle),
glm::vec3(0.0f, 1.0f, 0.0f));
model = glm::translate(model, glm::vec3(-8.0f, 2.0f,
0.0f));
model = glm::rotate(model, glm::radians(-20.0f),
glm::vec3(0.0f, 0.0f, 1.0f));
model = glm::scale(model, glm::vec3(3.0f, 3.0f, 3.0f));
model = glm::scale(model, glm::vec3(0.1f, 0.1f, 0.1f));
```

Kôd 6.1 Primjer svih transformacija s GLM bibliotekom

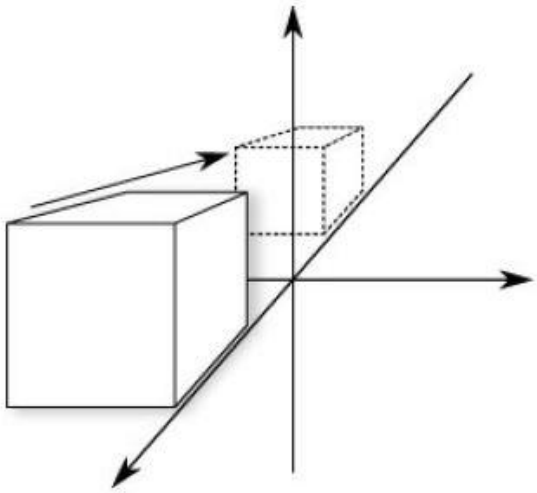
6.2.1. Translacija modela

GLM funkcija za translaciju modela:

```
glm::translate(model, glm::vec3(-8.0f, 2.0f, 0.0f));
```

U formuli (23) t predstavlja translaciju u prostoru. Govori za koliko će se model pomaknuti. v predstavlja poziciju s koje će se model translatirati. Kako to izgleda u praksi se može vidjeti na Slika 6.2.

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & t_x \\ 0.0 & 1.0 & 0.0 & t_y \\ 0.0 & 0.0 & 1.0 & t_z \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ 1.0 \end{bmatrix} = \begin{bmatrix} v_x + t_x \\ v_y + t_y \\ v_z + t_z \\ 1.0 \end{bmatrix} \quad (23)$$



Slika 6.2 Translacija modela po osi y za 10 [2]

6.2.2. Rotacija modela

GLM funkcija za rotaciju modela:

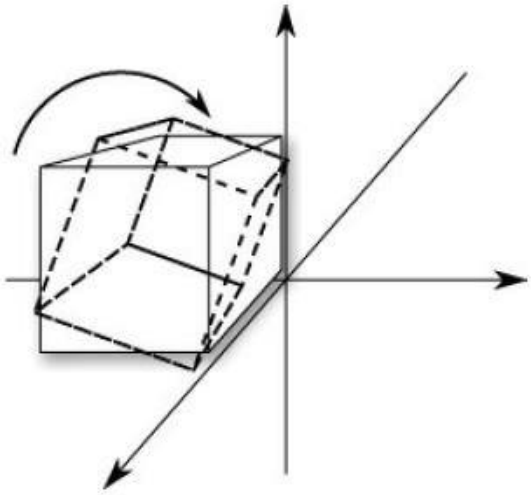
```
glm::rotate(model, glm::radians(-20.0f), glm::vec3(0.0f,
0.0f, 1.0f));
```

Objekt se može rotirati oko svih svojih osi. Formula za rotaciju ovisi oko koje osi se objekt rotira. U formulama $R_x(\theta), R_z(\theta)$ i $R_z(\theta)$ govori oko kojih će se osi rotirati, a θ koliki će biti kut rotacije. Formule su posebne za rotacije oko osi x (24), y (25) i z (26). Slika 6.3 je primjer rotacije modela.

$$R_x(\theta) = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & \cos\theta & \sin\theta & 0.0 \\ 0.0 & -\sin\theta & \cos\theta & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \quad (24)$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0.0 & -\sin\theta & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ \sin\theta & 0.0 & \cos\theta & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \quad (25)$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0.0 & 0.0 \\ \sin\theta & \cos\theta & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \quad (26)$$



Slika 6.3 Rotacija modela [2]

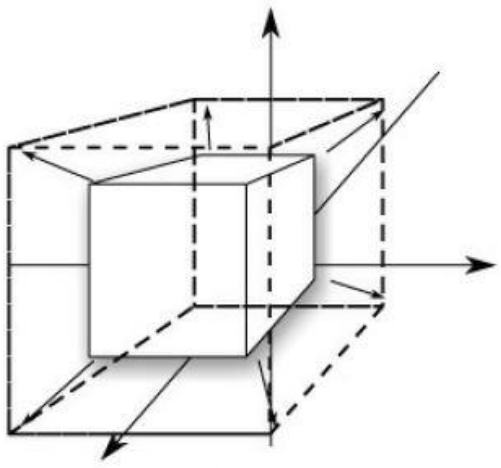
6.2.3. Skaliranje modela

GLM funkcija za skaliranje modela:

```
model = glm::scale(model, glm::vec3(3.0f, 3.0f, 3.0f));
```

Skaliranjem modela se mijenja oblik modela. Može se povećavati i smanjivati po sve tri koordinatne osi. Formula (27) transformaciju za skaliranje je najjednostavnija. $s_x, s_y \; i \; s_z$ predstavljaju jačinu skaliranja po svakoj osi. Slika 6.4 je vizualni primjer skaliranja modela.

$$\begin{bmatrix} s_x & 0.0 & 0.0 & 0.0 \\ 0.0 & s_y & 0.0 & 0.0 \\ 0.0 & 0.0 & s_z & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \tag{27}$$



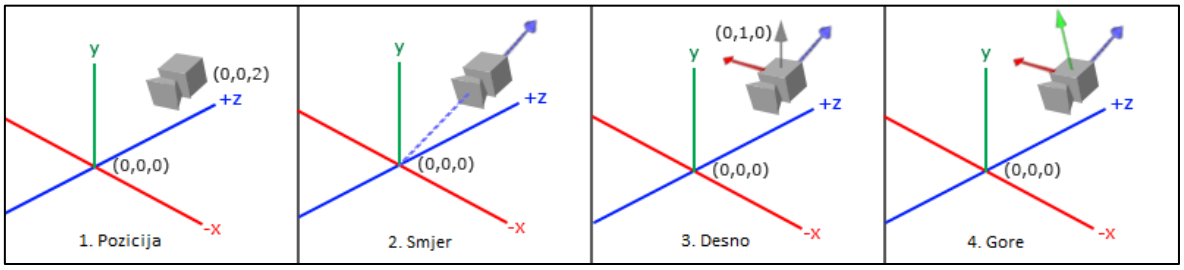
Slika 6.4 Skaliranje modela [2]

7. Kamera

U OpenGLu ne postoji koncept kamere, ali se može simulirati tako da se svi objekti u sceni pomiču u suprotnom smjeru od kretanja. Postoji više vrsta kamera, a govorit će se o izradi kamere iz perspektive prvog lica.

7.1. Kamera i prostor

Kada pričamo o kameri i prostoru onda pričamo o koordinatama točaka kako ih se vidi iz perspektive kamere koja je ishodište scene. Matrica pogleda transformira sve koordinate svijeta u koordinate pogleda koje su relativne poziciji i smjeru kamere. Za definiranje kamere trebamo njezinu poziciju u prostoru svijeta, smjeru u kojem gleda, vektor koji pokazuje desno i vektor koji pokazuje gore (iz kamere) kao što je vidljivo na Slika 7.1. Kreirat ćemo koordinatni sustav s 3 okomite jedinice osi s pozicijom kamere kao ishodištem. [5]



Slika 7.1 Postavljanje kamere [5]

Pozicija kamere je vektor u prostoru svijeta koji pokazuje na poziciju kamere. Kamera se postavi na neku proizvoljnu početnu poziciju:

```
glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
```

Pozitivna os z koja ide kroz ekran pokazuje prema vani tako da ako želimo da kamera ide prema nazad, treba je micati se niz pozitivnu z os.

Sljedeći vektor koji nam je potreban je smjer kamere koji govori u kojem smjeru pokazuje kamera. Prvotno će kamera pokazivati na ishodište scene (0,0,0). Oduzimanjem vektora pozicije kamere od vektora ishodišta scene dobiva se vektor smjera. Kamera pokazuje prema negativnom smjeru z, pa želimo da vektor smjera pokazuje prema kamerinoj pozitivnoj osi z. Oduzimanjem ishodišnog vektora scene od vektora pozicije kamere dobit će se vektor koji pokazuje prema pozitivnoj osi z kamere:

```
glm::vec3 cameraTarget = glm::vec3(0.0f, 0.0f, 0.0f);
glm::vec3 cameraDirection = glm::normalize(cameraPos -
cameraTarget);
```

Vektor smjera je varljiv naziv jer pokazuje u suprotnom smjeru od onoga na što je usmjeren.

Sljedeći vektor koji je potreban za kameru je vektor desno koji predstavlja pozitivnu os x prostora kamere. Da bi se dobio vektor desno prvo treba vektor gore koji pokazuje prema gore u prostoru svijeta. Zatim se napravi vektorski produkt vektora na vektoru gore i vektoru smjera iz drugog koraka. Rezultat je vektor okomit na oba vektora, a to je vektor koji pokazuje u smjeru pozitivne osi x:

```
glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f);
glm::vec3 cameraRight = glm::normalize(glm::cross(up,
cameraDirection));
```

Nakon što se izračunaju vektor osi x i vektor osi z potreban je još samo vektor koji pokazuje prema pozitivnoj y osi. Za taj vektor kamere treba izračunati vektorski produkt desnog vektora i vektora smjera:

```
glm::vec3 cameraUp = glm::cross(cameraDirection,
cameraRight);
```

U koordinatnom prostoru koji koristi 3 okomite osi može se napraviti matrica sa te 3 osi i vektor translacije te se može transformirati bilo koji vektor u taj koordinatni prostor tako da se pomnoži s ovom matricom. *LookAt* matrica (28) radi to, a sada kada imamo 3 okomite osi i vektor pozicije kako bismo definirali prostor kamere, možemo kreirati *LookAt* matricu

$$LookAt = \begin{bmatrix} R_x & R_y & R_z & 0.0 \\ U_x & U_y & U_z & 0.0 \\ D_x & D_y & D_z & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \cdot \begin{bmatrix} 1.0 & 0.0 & 0.0 & -P_x \\ 0.0 & 1.0 & 0.0 & -P_y \\ 0.0 & 0.0 & 1.0 & -P_z \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \quad (28)$$

- R je desni vektor
- U je vektor prema gore,
- D je vektor smjera
- P je vektor pozicije kamere

Vektor pozicije kamere je preokrenut tako što je postavljen na negativno jer želimo translatirati svijet u suprotnom smjeru od smjera u kojem se želimo kretati.

Koristeći *LookAt* matricu kao matricu pogleda, ona efektivno transformira sve koordinate svijeta u prostor pogleda. *LookAt* matrica onda radi što i kaže: kreira matricu pogleda koja gleda prema određenom smjeru.

GLM ima funkciju `lookAt` matrice. Funkciji je potrebno definirati poziciju kamere, ciljnu poziciju i vektor koji predstavlja vektor gore u prostoru svijeta, a za rezultat se dobije matrica pogleda:

```
glm::mat4 view;
view = glm::lookAt(glm::vec3(0.0f, 0.0f, 3.0f),
                   glm::vec3(0.0f, 0.0f, 0.0f),
                   glm::vec3(0.0f, 1.0f, 0.0f));
```

7.2. Kretanje kamere

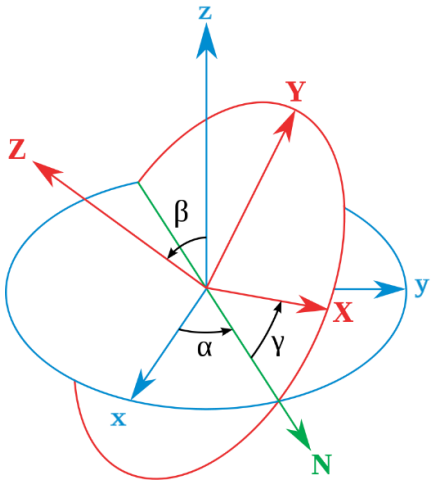
Kretanje kamere se odrađuje unosom preko tipkovnice i miša. Potrebno je prvo namjestiti sustav kamere, njezine varijable i postaviti `view` varijablu na novo početno stanje, a ovisno o pritisku tipke će se izračunati i postaviti nova vrijednost pozicije kamere. Sljedeći korak je osigurati brzinu kretanja koja će biti univerzalna na različitim računalima bez obzira na jačinu hardvera. Kada se to ne bi riješilo, na jačem hardveru bi kretanje s istim programom bilo brže nego na slabijem. Stoga je potrebno napraviti varijablu koja može neutralizirati brzinu pokreta. Prati se vrijeme potrebno za iscrtavanje sličice te se sprema u varijablu. Ako ta varijabla ima veću vrijednost znači da je prošlo više vremena od zadnjeg iscrtavanja, a manja vrijednost znači da je bilo potrebno manje vremena. Pokreti se množe s tom varijablom i dobiva se sličan ukupni pomak bez obzira na broj iscrtanih sličica. Naziv te varijable je najčešće *delta time* ili nešto slično. Kôd 7.1 prikazuje jednostavan slučaj.

```
float deltaTime = 0.0f;
float now = glfwGetTime();
deltaTime = now - lastFrame;
lastFrame = now;
void processInput(GLFWwindow *window)
{
      float cameraSpeed = 2.5f * deltaTime;
      ...
}
```

Kôd 7.1 Računanje `deltaTime` varijable

Većina 3D igara i aplikacija podržava gledanje u raznim smjerovima. Najčešće se gledanje u smjerovima odrađuje pomicanjem miša. Promjena smjera gledanja u kôdu znači promjenu vrijednosti varijable koja predstavlja vektor koji pokazuje iz kamere prema naprijed.

Za osnovni tip kamere koja može gledati horizontalno i vertikalno, to jest kameru pogleda iz prvog lica, potrebno je izračunati za koliko je potrebno prilagoditi kut i po kojoj osi. Slika 7.2 pokazuje kako se to može izračunati matematičkom tehnikom eulerovih kutova.

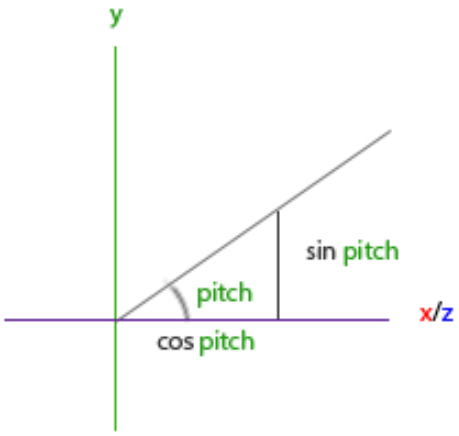


Slika 7.2 Eulerovi kutovi[18]

Prilikom vertikalnog okretanja (engl. *pitch*) moraju se prilagoditi vektori svih osi (29) po shemi sa Slika 7.3.

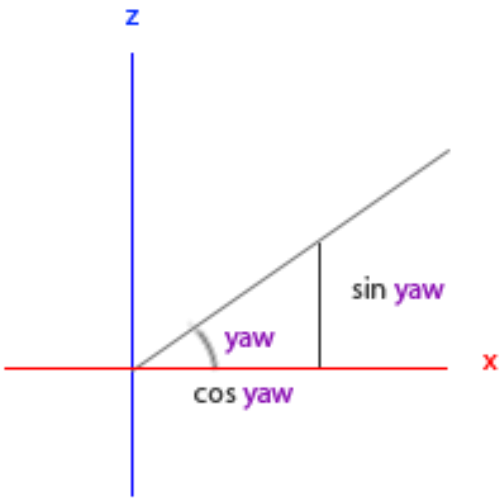
$$\begin{aligned} x &= \cos(pitch) \\ \mathrm{y} &= \sin(pitch) \\ z &= \cos(pitch) \end{aligned} \tag{29}$$

[18] https://en.wikipedia.org/wiki/Euler_angles



Slika 7.3 Računanje vertikalnog okretanja[19]

Za horizontalno okretanje (engl. *yaw*) potrebno je prilagoditi vektore po osi x i z (30) kao što se može vidjeti na Slika 7.4.



Slika 7.4 Računanje horizontalnog okretanja[19]

$$x = \cos(yaw)$$
$$z = \sin(yaw) \tag{30}$$

Da bi se dobio vektor smjera, potrebno je kombinirati okretanje u svim smjerovima, a to se može dobiti množenjem vektora po osima (31):

$$x = \cos(pitch) \cdot \cos(yaw)$$
$$\mathrm{y} = \sin(pitch) \tag{31}$$

[19] https://learnopengl.com/Getting-started/Camera

$$z = \cos(pitch) \cdot \sin(yaw)$$

Koristeći GLM biblioteku, dobivenu formulu se može jednostavno pretvoriti u kôd, s time što funkcije primaju kut u radijanima pa ih je potrebno pretvoriti koristeći funkciju `glm::radians()`:

```
direction.x = cos(glm::radians(pitch)) *
cos(glm::radians(yaw));
direction.y = sin(glm::radians(pitch));
direction.z = cos(glm::radians(pitch)) *
sin(glm::radians(yaw));
```

Izračune je potrebno povezati s mišem preko OpenGL-a. Za komunikaciju s mišem se koristi biblioteka GLFW koja će dohvaćati pomake funkcijom povratnog poziva (engl. *callback*). Obračun unosa se svodi na spremanje pozicije miša sa zadnje iscrtane sličice i uspoređivanje razlike udaljenosti s trenutnom. Dobivena razlika pomaka po osi x i y se prvo množi s varijablom `sensitivity` da se spriječi preveliki pomak, a zatim se ti rezultati pribrajaju varijabli kuta okretanja po osi x i drugoj varijabli za pomak po kutu osi y. Poželjno je ograničiti kut vertikalnog okretanja na manje od 90 stupnjeva da se kamera ne bi izvrtala, kao i postavljanje lokacije sredine prozora da kamera prilikom pokretanja ne napravi neželjeni pomak. Te funkcionalnosti su prikazane u Kôd 7.2.

```
void mouse_callback(GLFWwindow* window, double xpos, double
ypos);
glfwSetCursorPosCallback(window, mouse_callback);
void mouse_callback(GLFWwindow* window, double xpos, double
ypos)
{
      if(firstMouse)
      {
            lastX = xpos;
            lastY = ypos;
            firstMouse = false;
      }
      float xoffset = xpos - lastX;
      float yoffset = lastY - ypos;
      lastX = xpos;
      lastY = ypos;
      float sensitivity = 0.05;
      xoffset *= sensitivity;
      yoffset *= sensitivity;
```

```
        yaw = glm::mod( yaw + xoffset, 360.0f );;
        pitch = glm::mod(pitch + yoffset, 360.0f);
        if(pitch > 89.0f){pitch = 89.0f;}
        if(pitch < -89.0f){pitch = -89.0f;}
        glm::vec3 front;
        front.x = cos(glm::radians(yaw)) *
cos(glm::radians(pitch));
        front.y = sin(glm::radians(pitch));
        front.z = sin(glm::radians(yaw)) *
cos(glm::radians(pitch));
        cameraFront = glm::normalize(front);
}
```

Kôd 7.2 Postavljanje okretanja mišem [5]

8. Izrada praktičnog dijela projektnog zadatka

Tehnike, tehnologije i znanja iz prošlih poglavlja sam iskoristio za izradu praktičnog dijela projektnog zadatka koji iscrtava i animira modele te omogućava kretanje kroz prostor simulacijom kamere. To se sve rješava kombiniranjem korištenja biblioteka, kôda, matematičkih izračuna i direktnih poziva prema grafičkoj kartici. U ovom poglavlju će se na pojednostavljen način predočiti izrada bitnijih dijelova aplikacije. Svi primjeri kôda i slike u ovom poglavlju su primjeri iz demo aplikacije.

8.1. Postavljanje aplikacije i prozora

Osnovno postavljanje aplikacije se sastoji od dva dijela. Prvi dio je inicijalizacija GLEW biblioteke koja će se brinuti o ekstenzijama vezanim za platformu na kojoj se aplikacija koristi. Samo postavljanje GLEW biblioteke je jednostavno i radi se pozivanjem `glewInit()` funkcije vidljive u Kôd 8.1. Drugi dio je izrada prozora koji će prikazivati iscrtavanje modela u prostoru. Prozor će se izraditi pomoću GLFW okvira. Preko GLEW okvira je također postavljena verzija OpenGL-a koja se koristi, profil koji onemogućuje kompatibilnost sa starim verzijama OpenGL-a, a aktivirana je i kompatibilnost s novijim verzijama. Za prozor koji će se izraditi postavlja se visina i širina, ovisno o varijablama koje se postavljaju u konstruktoru. Povezuje se s međuspremnikom i postavlja kao kontekst koji će se iscrtati na prozoru. Pokazivač miša se sakriva, a iscrtavanje se postavlja preko cijelog prozora. Potrebne su i provjere koje će prilikom grešaka sigurno zatvoriti sve procese i dojaviti greške. Funkcije koje rade taj posao se mogu vidjeti kroz Kôd 8.1. Pozivom funkcije `InitialiseWindow()` u glavnom programu će se pojavit prozor.

```
void Window::GlfwStandardConfiguration()
{
      glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
      glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
      glfwWindowHint(GLFW_OPENGL_PROFILE,
GLFW_OPENGL_CORE_PROFILE);
      glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
      glewExperimental = GL_TRUE;
```

```
int Window::InitialiseWindow()
{
	bool initSuccess = glfwInit();
	{// error checking code...}
	GlfwStandardConfiguration();
	_window = glfwCreateWindow(_windowWidth,
_windowHeigth, "Name", NULL, NULL);
	if (!_window){//error checking code...}
	glfwGetFramebufferSize(_window, &_bufferWidth,
&_bufferHeigth);
	glfwMakeContextCurrent(_window);
	glfwSetInputMode(_window, GLFW_CURSOR,
GLFW_CURSOR_DISABLED);
	glEnable(GL_DEPTH_TEST);
	glViewport(0, 0, _bufferWidth, _bufferHeigth);
	GLenum error = glewInit();
	if (error != GLEW_OK){// error checking code...}}
	return 0;
}
```

Kôd 8.1 GLEW inicijalizacija i izrada prozora pomoću GLFW

8.2. Modeli i shaderi kao osnova iscrtavanja

Nakon što je prozor spreman, može se krenuti s učitavanjem i iscrtavanjem modela. Za aplikaciju su napravljena dva *shader* programa; *shader* vrha i *shader* fragmenta. Modeli su učitani uz pomoć biblioteke Assimp, ali logika za učitavanje je napravljena u klasi Model.

8.2.1. Shaderi aplikacije

Prvo je potrebno napraviti *shadere* koji će obrađivati modele na grafičkoj kartici. Njihov kôd se piše u posebnim datotekama, odvojenim od kôda same aplikacije. Učitava se prilikom pokretanja aplikacije. Napravljeni su *shader* vrha i *shader* fragmenta. *Shader* vrha iz aplikacije se može vidjeti u Kôd 8.2, a *shader* fragmenta u Kôd 8.4.

```
#version 330
layout (location = 0) in vec3 position;
layout (location = 1) in vec2 texture;
layout (location = 2) in vec3 normal;
out vec4 VertexColor;
```

```
out vec2 TextureCoord;
out vec3 Normal;
out vec3 FragPos;
uniform mat4 model;
uniform mat4 projection;
uniform mat4 view;

void main()
{
	gl_Position = projection * view * model *
vec4(position, 1.0);
	VertexColor = vec4(clamp(position, 0.0f, 1.0f),
1.0f);
	TextureCoord = texture;
	Normal = mat3(transpose(inverse(model))) * normal;
	FragPos = (model * vec4(position, 1.0)).xyz;
}
```

Kôd 8.2 *Shader* vrha

U Kôd 8.2 vrijednosti `position`, `texture` i `normal` predstavljaju redom vrijednosti pozicije, teksture i normala vrhova u prostoru. Tim vrijednostima se pristupa funkcijom `void glVertexAttribPointer()`. Kôd 8.3 prikazuje cijeli proces pristupa i postavljanja tih vrijednosti. Vrijednost `position` je potrebna da bi se GLSL funkcijom gl_`Position` izračunale pozicije modela u isječku prostora (engl. *clip space*) čije su koordinate od -1.0 do +1.0. Izračun se automatski prosljeđuje dalje kroz grafički cjevovod. Sve što se nalazi izvan tog prostora se uklanja i ne iscrtava se. `VertexColor, TextureCoord, Normal, FragPos i DirectionalLightSpacePos` redom predstavljaju vrijednosti boje, teksture, normala, pozicije fragmenta i pozicije osvjetljenja. To su `out` vrijednosti što znači da vrijednost prosljeđuju u daljnje *shadere*. U ovom slučaju će ga primiti *shader* fragmenta. [20]

```
//Korak 1
glGenVertexArrays(1, &_vao);
glBindVertexArray(_vao);
// stores indicies on graphics card
glGenBuffers(1, &_ibo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _ibo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indicies[0])*
numOfIndicies, indicies, GL_STATIC_DRAW);
```

[20] https://learnopengl.com/Getting-started/Coordinate-Systems

```
// stores vertices on graphics card
glGenBuffers(1, &_vbo);
glBindBuffer(GL_ARRAY_BUFFER, _vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(indicies[0])*
numOfVertices, vertices, GL_STATIC_DRAW);
//Korak 2
//shader (layout = 0), xyz values
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
sizeof(vertices[0]) * 8, 0);
glEnableVertexAttribArray(0);
//shader (layout = 1) uv values
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE,
sizeof(vertices[0]) * 8, (void*)(sizeof(vertices[0]) * 3));
glEnableVertexAttribArray(1);
//shader (layout = 2) nx, ny, nz values
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE,
sizeof(vertices[0]) * 8, (void*)(sizeof(vertices[0]) * 5));
glEnableVertexAttribArray(2);
//Korak 3
// unbinding, removing from graphics card
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

Kôd 8.3 Postavljanje `layout` vrijednosti

Korak 1 je faza pripreme. Kreiraju se spremnici i nizovi u koje će se vrijednosti vrhova spremiti, a zatim se vrhovi i njihov opis pohrane na memoriji grafičke kartice.

Korak 2 je faza povezivanja. Pokazivači se spoje na `layout` vrijednosti i aktiviraju njihovo korištenje.

Korak 3 je faza oslobađanja. Međuspremnici (engl. *buffers*) se prazne, a pokazivači oslobađaju.

```
#version 330
in vec4 vCol;
in vec2 TexCoord;
in vec3 Normal;
in vec3 FragPos;
in vec4 DirectionalLightSpacePos;
out vec4 colour;
```

```
struct Light
{
	vec3 colour;
	float ambientIntensity;
	float diffuseIntensity;
};
struct DirectionalLight
{
	Light base;
	vec3 direction;
};
struct Material
{
	float _specularIntensity;
	float _specularPower;
};
uniform DirectionalLight directionalLight;
uniform sampler2D theTexture;
uniform Material material;
uniform vec3 eyePosition;

vec4 CalcLightByDirection(Light light, vec3 direction)
{
// lighting calculations
...
	return (ambientColour + diffuseColour +
specularColour);
}
vec4 CalcDirectionalLight()
{
	return CalcLightByDirection(directionalLight.base,
directionalLight.direction);
}

void main()
{
	vec4 finalColour = CalcDirectionalLight();
	colour = texture(theTexture, TexCoord) * finalColour;
}
```

Kôd 8.4 Fragment *shader*

Fragment *shader* u Kôd 8.4 se prvenstveno bavi izračunom osvjetljenja po teksturama. Osvjetljenje je dodano u projekt za poboljšanje grafike, ali ne ulazi u sam sadržaj. Njegova varijabla `colour` je primjer jedine *out* varijable koju *shader* automatski preuzima kao vrijednost boje i prosljeđuje dalje u grafički cjevovod. *Shadere* je potrebno povezati s grafičkom karticom, glavnim programom i modelima. To je odrađeno u Shader klasi, a važniji dio klase se može vidjeti u pojednostavljenom primjeru Kôd 8.5. `CompileShader()` funkcija se bavi povezivanjem *shader* programa i njegovih varijabli s grafičkom karticom. Za to se koriste OpenGL naredbe `glCreateProgram()` i `glLinkProgram()` koje zajedno na `_shaderID` varijablu povežu *shadere* i omoguće pristup i korištenje programa. `AddShader()` funkcijom se kôd pročitan iz *shader* datoteka upisuje u program prije nego što ga se poveže na *shader* jezgre grafičke kartice.

```
void Shader::CompileShader(const char * vertexCode, const
char * fragmentCode)
{
	_shaderID = glCreateProgram();
	AddShader(_shaderID, vertexCode, GL_VERTEX_SHADER);
	AddShader(_shaderID, fragmentCode,
GL_FRAGMENT_SHADER);
	glLinkProgram(_shaderID);
	_uniformProjection = glGetUniformLocation(_shaderID,
"projection");
// code for setting other uniforms
...
}
void Shader::AddShader(GLuint theProgram, const char *
shaderCode, GLenum shaderType)
{
	GLuint theShader = glCreateShader(shaderType);
	const GLchar* theCode[1];
	theCode[0] = shaderCode;
	GLint codeLength[1];
	codeLength[0] = strlen(shaderCode);
	glShaderSource(theShader, 1, theCode, codeLength);
	glCompileShader(theShader);
	glAttachShader(theProgram, theShader);
	return;
}
```

Kôd 8.5 Funkcije *shader* klase

8.2.2. Učitavanje modela iz datoteka

Modeli se učitavaju uz pomoć biblioteke Assimp koja omogućava univerzalno učitavanje različitih formata modela, ali potrebno je ustrojiti logiku samog učitavanja. Kôd 8.6 prikazuje klasu Model u kojoj je implementirana logika učitavanja napravljena po dizajnu Assimp biblioteke sa Slika 3.1. Kôd u primjeru je pojednostavljen za učitavanje modela bez grešaka, ali kôd u aplikaciji ima i kôd za obradu grešaka i nedostataka modela. Učitavanje tekstura je napravljeno u odvojenoj Texture klasi. Najviše se svodi na učitavanje korištenjem jednostavne biblioteke stb[21] i njezine funkcije `stbi_load()` koja odrađuje većinu posla.

```
void Model::LoadModel(const std::string & fileName,
std::string subfolder)
{
	Assimp::Importer importer;
	const aiScene * scene = importer.ReadFile(fileName,
aiProcess_Triangulate | aiProcess_FlipUVs |
aiProcess_GenSmoothNormals |
aiProcess_JoinIdenticalVertices);
	_subfolder = "/" + subfolder + "/";
	LoadNode(scene->mRootNode, scene);
	LoadMaterials(scene);
}

void Model::RenderModel()
{
	for (size_t i = 0; i < _meshes.size(); i++)
	{
		unsigned int materialIndex = _meshToTexure[i];
		if (materialIndex < _textures.size() &&
_textures[materialIndex] != nullptr)
		{_textures[materialIndex]->UseTexture();}
		_meshes[i]->RenderMesh();
	}
}
void Model::LoadNode(aiNode * node, const aiScene * scene)
{
	for (size_t i = 0; i < node->mNumMeshes; i++)
	{LoadMesh(scene->mMeshes[node->mMeshes[i]], scene);}
```

[21] https://github.com/nothings/stb

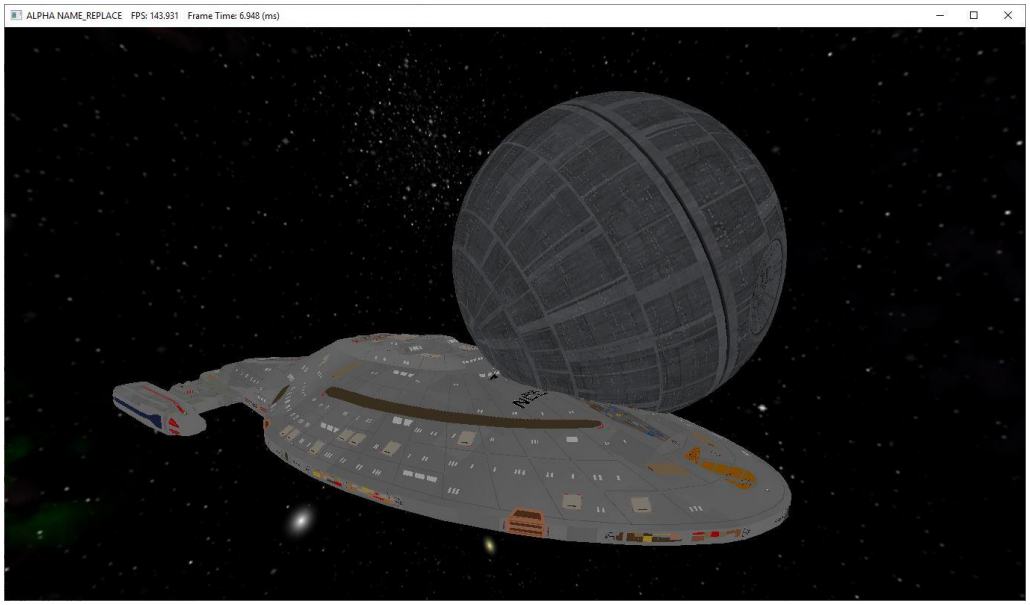
```
	for (size_t i = 0; i < node->mNumChildren; i++)
	{LoadNode(node->mChildren[i], scene);}
}

// connecting vertices with textures
void Model::LoadMesh(aiMesh * mesh, const aiScene * scene)
{
	 //code..
}

void Model::LoadMaterials(const aiScene * scene)
{
	_textures.resize(scene->mNumMaterials);

	for (size_t i = 0; i < scene->mNumMaterials; i++)
	{
		aiMaterial * material = scene->mMaterials[i];
		_textures[i] = nullptr;
		// looking for diffuse texture
		if (material-
>GetTextureCount(aiTextureType_DIFFUSE))
		{
			aiString path;
			if (material-
>GetTexture(aiTextureType_DIFFUSE, 0, &path) == AI_SUCCESS)
			{
				int index =
std::string(path.data).rfind("\\");
				std::string fileName =
std::string(path.data).substr(index + 1);
				std::string texturePath =
std::string("Textures/") + _subfolder + fileName;
				_textures[i] = new
Texture(texturePath.c_str());
	}
}
```

Kôd 8.6 Klasa Model za učitavanje modela

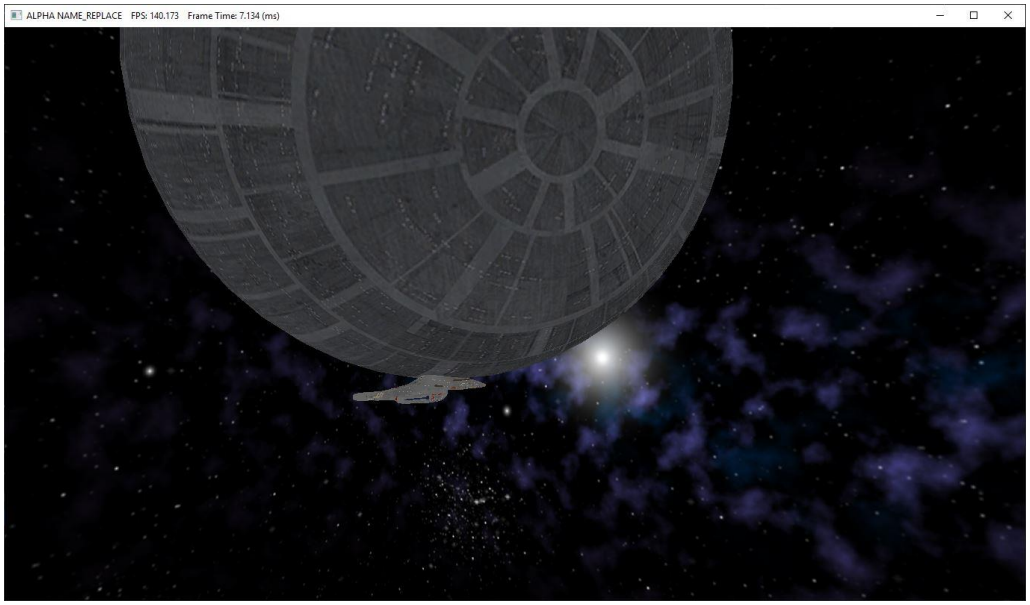
Ovim dijelom aplikacije omogućilo se učitavanje i iscrtavanje modela, vidljivo na **Pogreška! Izvor reference nije pronađen.**.

8.3. Prilagodba i animacija modela

Učitani modeli bez prilagodbe su nepomični i često ne odgovaraju veličinom i početnom pozicijom kao što se može vidjeti na Slika 8.1, gdje su modeli u koliziji i neproporcionalni.



Slika 8.1 Modeli[22] [23] učitani bez prilagodbe



Slika 8.2 Prilagođene veličine modela[22] [23]

Pozivanjem funkcije `glm::scale()` na model broda i postavljanjem vrijednosti na jednu desetinu dobiju se željene veličine modela:

```
model = glm::scale(model, glm::vec3(0.1f, 0.1f, 0.1f));
```

[22] https://free3d.com/3d-model/voyager-ncc-74656-49099.html
[23] https://free3d.com/3d-model/puo-4037-34535.html

Kao što se vidi na Slika 8.2, veličine su dobre, ali pozicije nisu, tako da je sljedeći korak translatirati model broda pozivom funkcije `glm::translate()` prvo na svemirski brod, a zatim na zvijezdu smrti:

```
model = glm::translate(model, glm::vec3(-8.0f, 2.0f, 0.0f));
model = glm::translate(model, glm::vec3(10.0f, -10.0f, -
3.0f));
```

Slika 8.3 Translacija na modelima[22] [23]

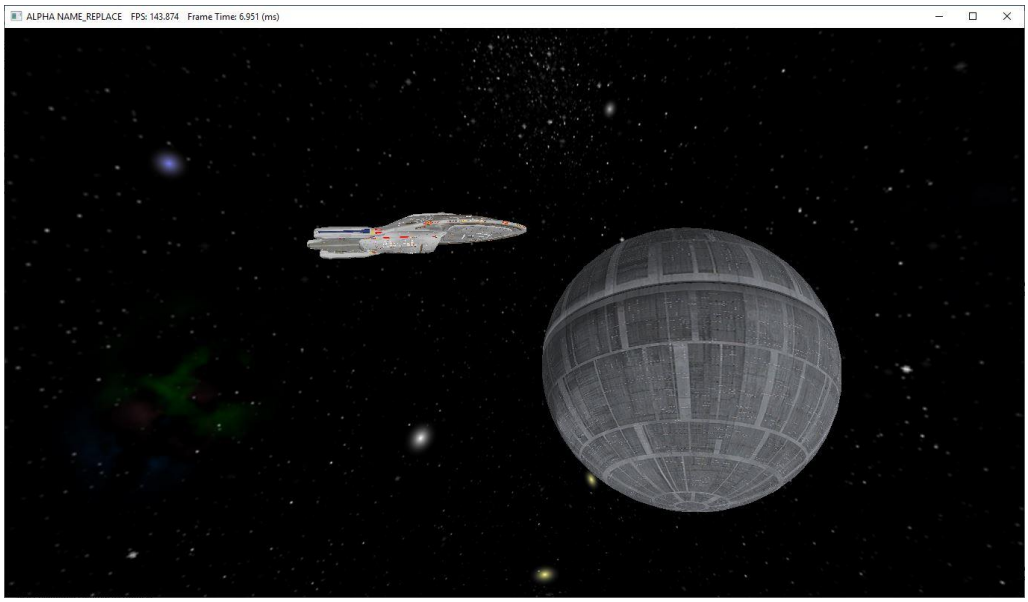
Dobivena udaljenost vidljiva na Slika 8.3 je prihvatljiva pa je vrijeme za simulaciju kretanja. Funkcija `RenderModels()` iz Kôd 8.7 pokazuje kako se u aplikaciji napravila simulacija kretanja. Funkciju `RenderModels()` se poziva prilikom svakog iscrtavanja, a ona prilagođava poziciju modela kako bi modeli izgledali kao da se pomiču.

```
void RenderModels(const GLuint &uniformModel, const GLuint
&uniformSpecularIntensity, const GLuint &uniformShininess)
{
	glm::mat4 model{ 1.0f };
	startingAngle += 0.1f;
	if (startingAngle > 360.0f)
	{startingAngle = 0.1f;}

	// svemirski brod
	model = glm::mat4{ 1.0f };
	model = glm::translate(model, glm::vec3(-8.0f, 2.0f,
0.0f));
```

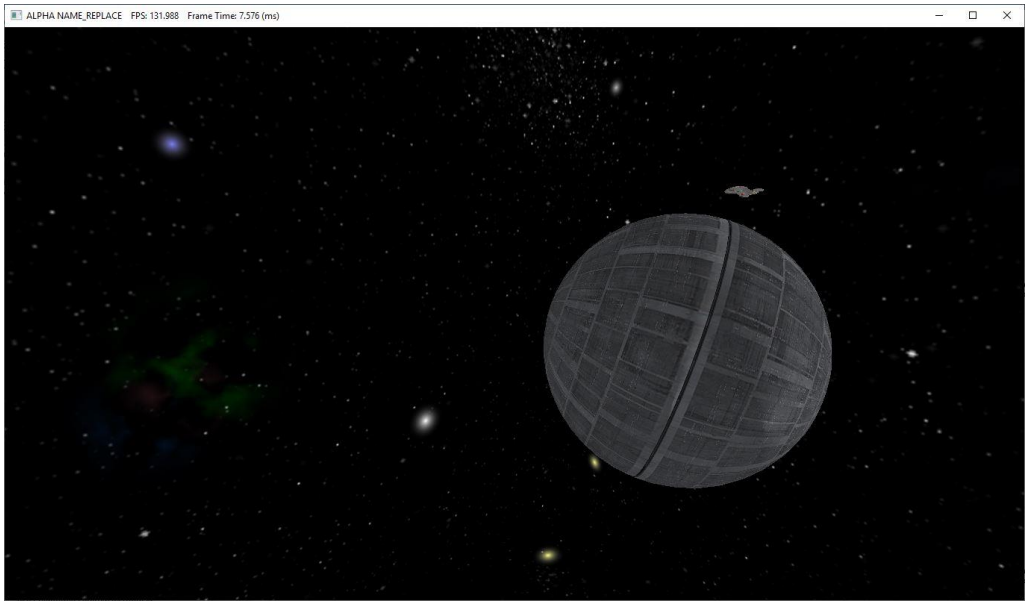
```
		model = glm::rotate(model,
glm::radians(+startingAngle), glm::vec3(0.0f, 1.0f, 0.0f));
		model = glm::translate(model, glm::vec3(-8.0f, 2.0f,
0.0f));
		model = glm::rotate(model, glm::radians(-20.0f),
glm::vec3(0.0f, 0.0f, 1.0f));
		model = glm::scale(model, glm::vec3(0.1f, 0.1f,
0.1f));
		glUniformMatrix4fv(uniformModel, 1, GL_FALSE,
glm::value_ptr(model));
		shinyMaterial.UseMaterial(uniformSpecularIntensity,
uniformShininess);
		testModel.RenderModel();

		// zvijezda smrti
		model = glm::mat4{ 1.0f };
		model = glm::translate(model, glm::vec3(10.0f, -
10.0f, -3.0f));
		model = glm::rotate(model, glm::radians(20.0f),
glm::vec3(0.0f, 0.0f, 1.0f));
		model = glm::rotate(model,
glm::radians(startingAngle), glm::vec3(0.0f, 0.5, 0.0f));
		model = glm::scale(model, glm::vec3(3.0f, 3.0f,
3.0f));
		glUniformMatrix4fv(uniformModel, 1, GL_FALSE,
glm::value_ptr(model));
		shinyMaterial.UseMaterial(uniformSpecularIntensity,
uniformShininess);
		moon.RenderModel();
}
```

Kôd 8.7 Funkcija iscrtavanja i simulacije pokreta modela

Slika 8.4 i Slika 8.5 prikazuju kako se modeli pomiču tijekom rada aplikacije. Zvijezda smrti se vrti oko svoje osi, a svemirski brod kruži u neposrednoj blizini:



Slika 8.4 Simulacija pokreta modela[22] [23] 1



Slika 8.5 Simulacija pokreta modela[22] [23] 2

8.4. Implementacija kamere

Poglavlje 7 se bavi detaljnim opisom rada kamere tako da će se ovdje prikazati samo konkretna implementacija kamere u aplikaciji. Prvo je potrebno pozvati konstruktor kamere da se postavi početna pozicija te izračunati projekciju:

```
camera = Camera(glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f,
1.0f, 0.0f), -60.0f, 0.0f, 5.0f, 0.5f);
```

```
glm::mat4 projection = glm::perspective(45.0f,
(GLfloat)mainWindow.GetBufferWidth() /
mainWindow.GetBufferHeight(), 0.1f, 100.0f);
```

Svakim iscrtavanjem se poziva metoda `Render()` iz Kôd 8.8 koja će dobiti i proslijediti sve odrađene izračune prostora i pozicije za kameru, da bi se prilikom novog iscrtavanja promijenila pozicija pogleda.

```
void Render(GLuint &uniformModel, GLuint
&uniformProjection, GLuint &uniformView, GLuint
&uniformEyePosition, GLuint &uniformSpecularIntensity,
GLuint &uniformShininess, glm::mat4 projectionMatrix,
glm::mat4 viewMatrix)
{
	glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
	shaderList[0]->UseShader();
	uniformModel = shaderList[0]->GetModelLocation();
	// getting other uniforms
	{ // code...}
	shaderList[0]->SetDirectionalLight(&mainLight);
	glUniformMatrix4fv(uniformProjection, 1, GL_FALSE,
glm::value_ptr(projectionMatrix));
	glUniformMatrix4fv(uniformView, 1, GL_FALSE,
glm::value_ptr(camera.CalculateViewMatrix()));
	glUniform3f(uniformEyePosition,
camera.GetCameraPosition().x, camera.GetCameraPosition().y,
camera.GetCameraPosition().z);
}
```

Kôd 8.8 Funkcija `RenderModels()`

Klasa Camera se primarno bavi računanjem vektora kamere, a nakon što se pozove `glfwPollEvents()`, funkcija koja omogućava dohvaćanje aktivaciju tipki tipkovnice i pokreta miša, u suradnji s klasom Window iščitava aktivirane tipke i pokrete te ovisno o tome izračunava potrebnu prilagodbu lokacije i kuta prikaza iz kojeg će se iscrtavati scena u prozoru. Bitne funkcije koje to obavljaju su prikazane u Kôd 8.9.

```
void Camera::KeyboardControl(bool * keys, GLfloat
deltaTime)
{
	GLfloat velocity = _moveSpeed * deltaTime;
```

```
	if (keys[GLFW_KEY_W])
	{_position += _front * velocity;}
	if (keys[GLFW_KEY_S])
	{_position -= _front * velocity;}
	if (keys[GLFW_KEY_A])
	{_position -= _right * velocity;    }
	if (keys[GLFW_KEY_D])
	{_position += _right * velocity;    }

void Camera::MouseControl(GLfloat xChange, GLfloat yChange)
{
	xChange *= _turnSpeed;
	yChange *= _turnSpeed;
	_yaw += xChange;
	_pitch += yChange;
	Set_FrontRightUp();
}
glm::vec3 Camera::GetCameraPosition()
{return glm::normalize(_front);}
glm::vec3 Camera::GetCameraDirection()
{return glm::normalize(_front);}
glm::mat4 Camera::CalculateViewMatrix()
{return glm::lookAt(_position, _position + _front, _up);}
void Camera::Set_FrontRightUp()
{
	_front.x = cos(glm::radians(_yaw)) *
cos(glm::radians(_pitch));
	_front.y = sin(glm::radians(_pitch));
	_front.z = sin(glm::radians(_yaw)) *
cos(glm::radians(_pitch));
	_front = glm::normalize(_front); // normalizing
	_right = glm::normalize(glm::cross(_front,
_worldUp));
	_up = glm::normalize(glm::cross(_right, _front));
}
void Window::InitializeKeyboardKeys()
{ // loading and setting keyboard keys}

void Window::CreateEnableCallbacks()
{
	glfwSetKeyCallback(_window, GetKeyboardInput);
```

```
	glfwSetCursorPosCallback(_window, GetMouseInput);
}

void Window::GetKeyboardInput(GLFWwindow * window, int key,
int code, int action, int mode)
{ // handling keyboard input...}

void Window::GetMouseInput(GLFWwindow * window, double
xPosition, double yPosition)
{ // handling keyboard input...}
```

Kôd 8.9 Klasa Camera i funkcije klase Window

8.5. Povezivanje u glavni program

Dijelove programa je potrebno povezati, funkcije pozvati, *shadere* i modele učitati iz datoteka, varijable pripremiti i inicijalizirati, a sve to vrtiti kroz glavnu petlju za iscrtavanje. Klasa Program koja to zaokružuje u cjelinu je prikazana u Kôd 8.10. `while (mainWindow.GetWindowShouldClose() == false)` funkcija iz tog kôda predstavlja glavnu petlju koja će stalno pokretati iscrtavanje dok se prozor ne zatvori.

```
void CreateShaders() {// reading and loading from files}
void RenderModels(const GLuint &uniformModel, const GLuint
&uniformSpecularIntensity, const GLuint &uniformShininess)
{     // kôd...}
void Render(GLuint &uniformModel, GLuint
&uniformProjection, GLuint &uniformView, GLuint
&uniformEyePosition, GLuint &uniformSpecularIntensity,
GLuint &uniformShininess, glm::mat4 projectionMatrix,
glm::mat4 viewMatrix)
{// kôd...}
int main(int argc, char** argv)
{
	Window mainWindow = Window(1366, 768);
	mainWindow.InitialiseWindow();
	CreateShaders();
	camera = Camera(glm::vec3(0.0f, 0.0f, 0.0f),
glm::vec3(0.0f, 1.0f, 0.0f), -60.0f, 0.0f, 5.0f, 0.5f);
	GLuint uniformProjection = 0, uniformModel = 0,
uniformView = 0, uniformEyePosition = 0,
```

```
            uniformSpecularIntensity = 0, uniformShininess
= 0;
      glm::mat4 projection = glm::perspective(45.0f,
(GLfloat)mainWindow.GetBufferWidth() /
mainWindow.GetBufferHeight(), 0.1f, 100.0f);
      plainTexture = Texture("Textures/plain.png");
      plainTexture.LoadTextureAlpha();
      shinyMaterial = Material(4.0f, 256);
      voyager.LoadModel("Models/voyager.obj", "Voyager");
      moon = Model();
      moon.LoadModel("Models/Death_Star.obj", "DeathStar");

      while (mainWindow.GetWindowShouldClose() == false)
      {
            Render(uniformModel, uniformProjection,
uniformView, uniformEyePosition, uniformSpecularIntensity,
uniformShininess, projection,
camera.CalculateViewMatrix());
            RenderModels(uniformModel,
uniformSpecularIntensity, uniformShininess);
            glUseProgram(0);
            mainWindow.SwapBuffers();
            glfwPollEvents();
      camera.KeyboardControl(mainWindow.GetKeyboardKeys(),
deltaTime);
      camera.MouseControl(mainWindow.GetXChange(),
mainWindow.GetYChange());
      }
      return 0;
}
```

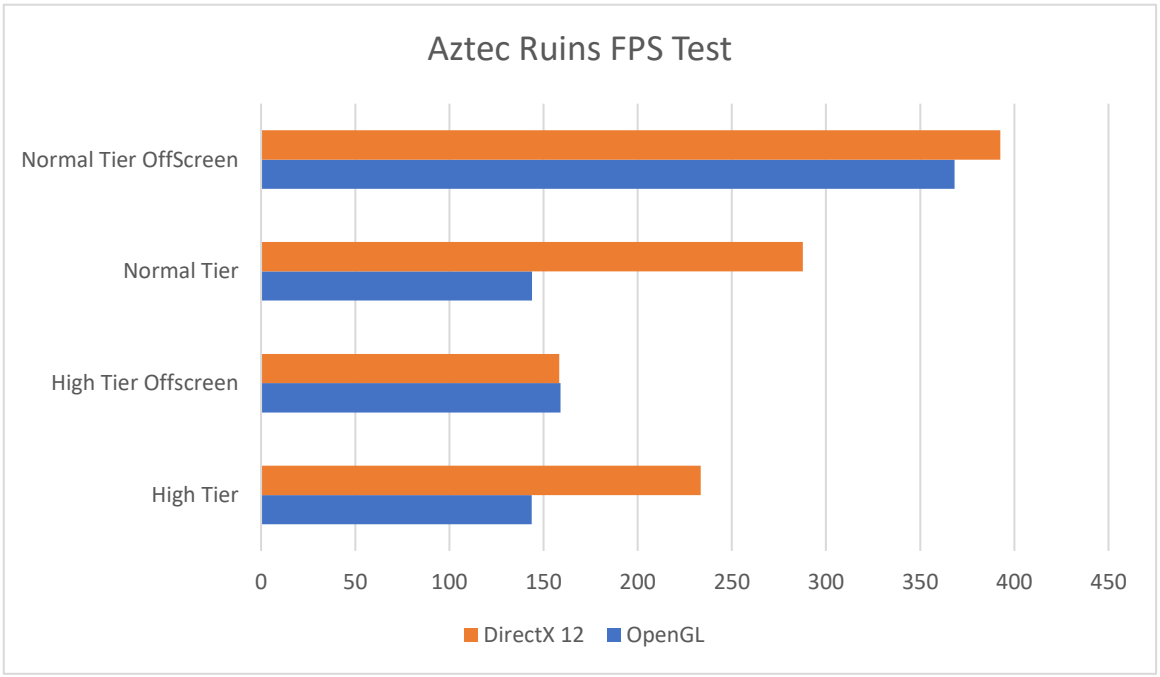
Kôd 8.10 Glavni program

9. Analiza i usporedba postojećih grafičkih API-a

U ovom poglavlju će se usporediti performanse, međusobni odnos, trenutno stanje i budućnost OpenGL-a s nekim od poznatijih i popularnijih grafičkih programskih sučelja. Usporedno testiranje grafičkih API-a se ne radi često, a razlog tome je to što takva testiranja ovise o puno parametara, kao i samom programu koji se pokreće, pa ne mogu biti potpuno precizni, ali mogu dati solidnu sliku stanja. Program izrađen uz ovaj rad je baziran na OpenGL API-u, a kroz analize i usporedbe će se prikazati koje su alternative za OpenGL i kakve se performanse mogu očekivati korištenjem drugih popularnih grafičkih API-a. Obzirom da bi za testiranje ovog rada s različitim grafičkim API-ima bilo potrebno znati DirectX, Vulkan i Metal tehnologije koje prilično razlikuju od OpenGL API-a te izraditi kompletne nove implementacije, koristit će se tuđi i stručniji primjeri. Jedino će se u slučaju usporedbe OpenGL-a i DirectX-a zbog nedostatka stručnih usporedbi koristiti aplikacija GFXBench koja će testirati svoj program s implementacijom i OpenGL-a i DirectX-a na grafičkoj kartici mog računala.

9.1. Usporedba OpenGL-a i DirectX-a

DirectX je kolekcija programskih sučelja primarno ciljanih za izradu multimedije s naglaskom na izradu igara za Windowse i Xbox One konzole. Najnovija verzija je 12.

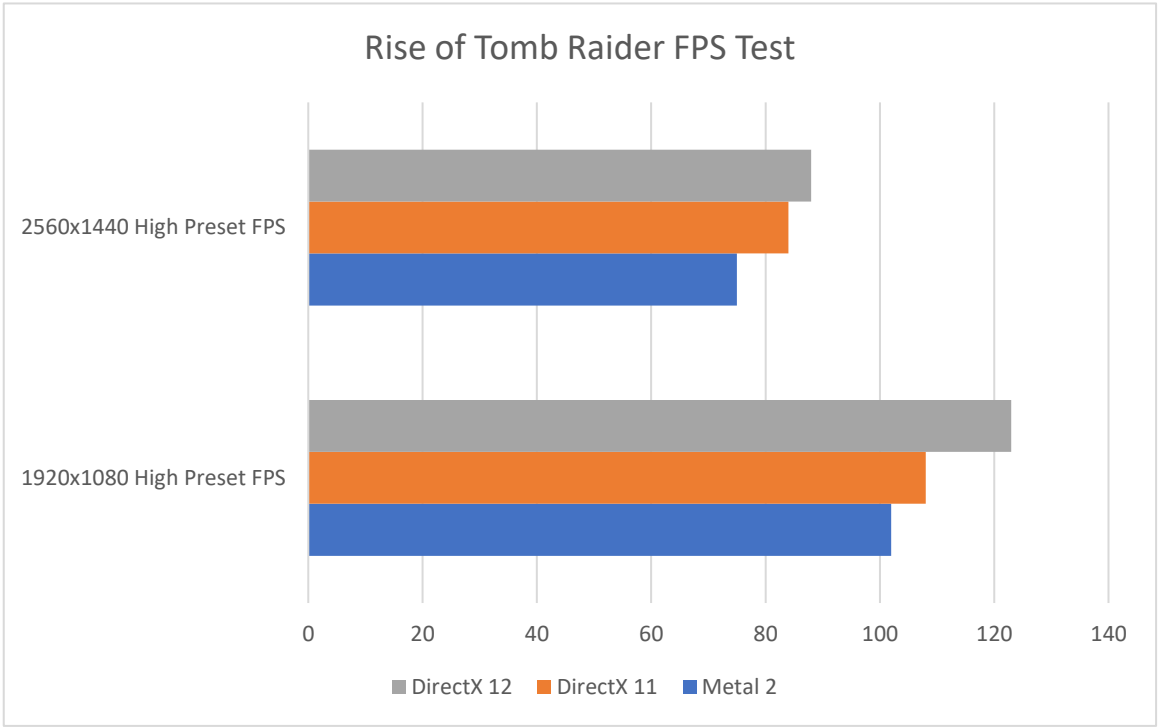


Slika 9.1 Rezultat testiranja OpenGL-a i DirectX-a 12 na GFXBench aplikaciji

Kôd za DirectX se piše u programskim jezicima C# i C++.[24] Testiranje je napravljeno koristeći program Aztec Ruins GFXBench-a na grafičkoj kartici Nvidia GeForce GTX 1070 na mojem osobnom računalu zbog nedostatka prikladnih testova. Po rezultatima sa Slika 9.1, jasno je da se pod većim opterećenjem OpenGL teško može mjeriti s DirectXom 12 po pitanju performansi. Prednost OpenGL-a nad DirectX-om je što se nalazi na mnogim platformama poput Linux-a, Windows-a, macOS-a i android-a, dok je DirectX kompatibilan samo s Windows operativnim sustavom, a verzija 12 samo sa Windows-om 10 i konzolom Xbox One. DirectX nije svestran kao OpenGL tako da se ne očekuje da bi ga mogao zamijeniti bez obzira što DirectX ima bolje performansame na Microsoftovim platformama.

9.2. Usporedba OpenGL-a i Metala

Metal je grafičko sučelje koje je razvila tvrtka Apple. Cilj tvrtke Apple je izrada grafičkog sučelja koje može razvijati specifično za svoje proizvode kao što to radi Microsoft s DirectX-om. S Metal-om se želi na iOS, macOS i tvOS[7] donijeti kvaliteta jednaka DirectX-u i Vulkan-u. Programski jezici za razvoj Metal-a su Swift i Objective C. [25]

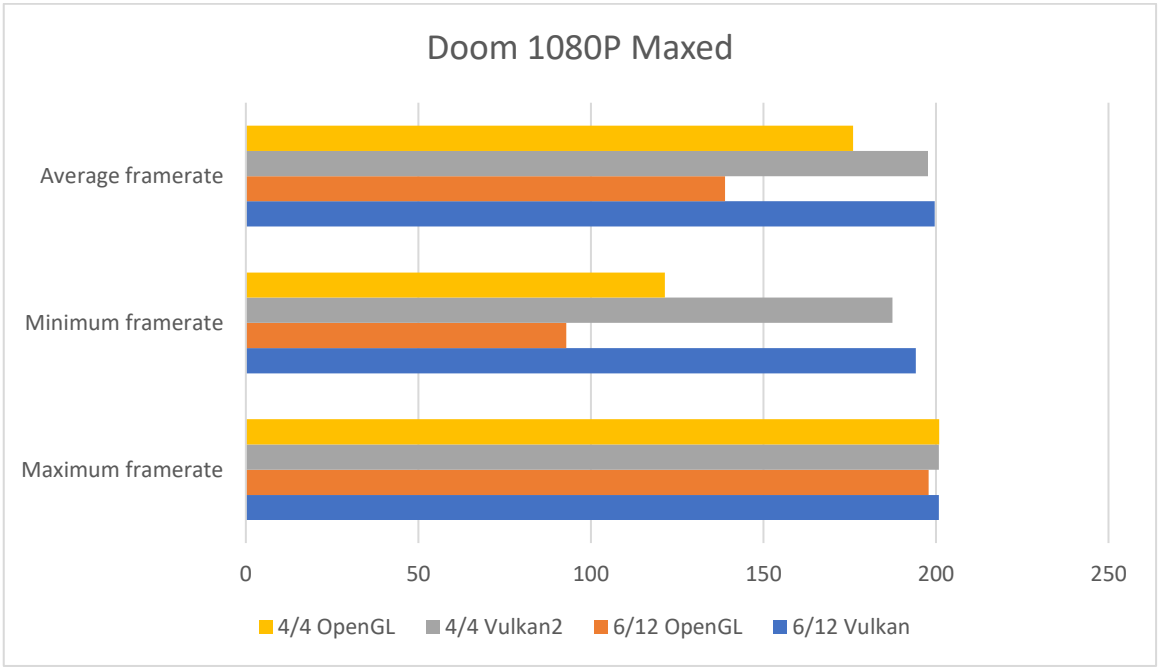


Slika 9.2 Rezultat testiranja igre Rise of Tomb Raider na FullHD i QHD rezoluciji[26]

[24] https://en.wikipedia.org/wiki/DirectX
[25] https://en.wikipedia.org/wiki/Metal_(API)
[26] https://barefeats.com/directx_versus_metal_2.html

Trenutno nema izravne usporedbe OpenGL-a i Metala pa se prema Slika 9.2 usporedbe s DirectX-om 12 se da zaključiti da su performanse Metala negdje na sredini između DirectX-a i OpenGL-a. Direktna usporedba niti nije toliko važna jer Apple je 2018. godine objavio potpuni prelazak na Metal i prekid podrške za OpenGL. OpenGL će biti korišteniji grafički API još dugo vremena jer Metal je ograničen na specifičnu grupu operativnih sustava.

9.3. Usporedba OpenGL-a i Vulkan-a



Slika 9.3 Rezultati testiranja igre Doom na FullHD rezoluciji[27]

OpenGL i Vulkan su testirani na igri Doom koristeći grafičku karticu Nvidia GeForce GTX 1080. Na Slika 9.3 može se uočiti je da je Vulkan puno stabilniji u radu od OpenGL-a. Vulkan-ov prosječni broj sličica u sekundi (engl. *frames per second*, skraćeno FPS) je skoro identičan maksimalnom broju dok je minimalni broj sličica u sekundi blizu prosječnom. OpenGL ima identičan maksimalan FPS, ali padovi su vrlo veliki. Minimalni FPS se spušta i na niskih 93 FPS-a dok je prosječni FPS znatno manji od Vulkan-ovog. Razlika je od 20 do 60 FPS-a. To je očekivano jer je Vulkan nova generacija grafičkog API-a Khronos grupe, konzorcija koji razvija OpenGL.[28] Očekuje se da će Vulkan polako zamjenjivati OpenGL u modernim igrama, ali za android platformu, desktop grafičke aplikacije i *indie* igre će se vjerojatno zadržati još dosta vremena zbog svoje robusnosti i raširenosti.

[27] http://www.redgamingtech.com/how-much-better-is-performance-with-modern-apis-directx-12-vs-directx-11-opengl-vs-vulkan/
[28] https://www.khronos.org/vulkan/

Zaključak

Programiranje 3D grafike je jedno od najkompliciranijih područja programiranja jer uključuje više različitih segmenata i osobito mnogo matematike koja postaje sve zahtjevnija što se dublje ulazi u naprednije tehnike 3D prikaza. Također je jedno od najzanimljivijih područja programiranja jer modelima daje život kroz animaciju, boje i nijanse, a to je početak nastanka svih mogućih svjetova računalnih igara.

Kroz ovaj proces sam naučio puno matematike i specifičnosti komuniciranja s grafičkom karticom koje je prilično složeno budući da OpenGL radi po principu stroja stanja. Matematika nije toliko teška sama po sebi, ali kada se koristi za izračune u 3D prostoru stvari se izuzetno zakompliciraju jer se broj kombinacija brzo povećava. *Shaderi* su izazovni jer su oni najbliži doticaj rada s grafičkom karticom. Potrebno ih je precizno izraditi, inače niti jedan dio aplikacije neće funkcionirati. Imaju i svoje specifičnosti koje omogućavaju postizanje vrlo impresivnog izgleda modela, pogotovo na današnjim grafičkim karticama. Izradom transformacija modela i kamere aplikacija se počinje kretati u smjeru računalnih igara. Kamera je najzanimljivija za izradu jer, osim što omogućava kretanje kroz prostor, pokazuje koliko je transformacija potrebno da se napravi iluzija 3D grafike na 2D ekranu.

OpenGL nije više jedan od nadmoćnijih grafičkih API-a. Iako je svestran i može se koristiti na većini platformi, najmodernije aplikacije, softver i hardver polako prelaze na Vulkan API, njegov moderni nasljednik. Zahvaljujući svojoj rasprostranjenosti održati će se još dulje vrijeme, ali je upitno koliko dugo će se još nastaviti njegov razvoj. Vulkan API je još „mlad“, ali mogao bi u bliskoj budućnosti postati standard koji će biti neophodno dobro poznavati.

Znanje usvojeno pri izradi za mene je najvredniji dio rada i važan prvi korak u smjeru karijere u industriji igara. OpenGL se pokazao kao odličan uvod u grafičko programiranje.

Popis kratica

AF	*Anisotropic filtering*	Anizotropno filtriranje
API	*Application programming interface*	Aplikacijsko programsko sučelje
CTO	*Chief technology officer*	Glavni tehnički direktor
FPS	*Frames per second*	Broj sličica u sekundi
SDK	*Software development kit*	Komplet za razvoj softvera

Popis slika

Popis tablica

Popis kôdova

Literatura

[1] KESSENICH, J, SELLERS, G, SHREINER, D, *OpenGL Programming Guide : The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V*: Addison-Wesley Professional, 2016.

[2] WRIGHT, R, S, SELLERS, G, HAEMEL, N, *OpenGL Superbible : Comprehensive Tutorial and Reference:* Addison-Wesley Professional, 2016.

[3] LENGYEL, E, *Mathematics for 3D Game Programming and Computer Graphics Third Edition:* Course Technology, 2012.

[4] BAILEY, M, CUNNINGHAM, S, *Graphics Shaders: Theory and Practice Second Edition:* CRC Press, 2012.

[5] LEARN OPENGL, Camera, https:// learnopengl.com

[6] KHRONOS, OpenGL overview, https://www.khronos.org

[7] DEVELOPER APPLE, Metal 2, https://developer.apple.com/metal

[8] WOLF, D, *OpenGL 4 Shading Language Cookbook Second Edition:* Packt Publishing Ltd., 2013.

[9] GREGORY, J, *Game Engine Architecture Second Edition:* CRC Press, 2014.

Prilog


ALGEBRA
VISOKO
UČILIŠTE
Prikaz i pokret 3D modela u prostoru
Datum: 18. 02. 2019.